

Orbeon PresentationServer (OPS)

User Guide

The Orbeon PresentationServer Reference

Table of Contents

1. Getting Started

1.1. Welcome

- 1.1.1. Introduction
- 1.1.2. Purpose of the User Guide
- 1.1.3. Prerequisites

1.2. Changes in Version 3.0

- 1.2.1. Introduction
- 1.2.2. Known Limitations of OPS 3.0
- 1.2.3. XForms
- 1.2.4. Page Flow
- 1.2.5. XSLT processor
- 1.2.6. XQuery processor
- 1.2.7. XHTML Support
- 1.2.8. SQL Processor
- 1.2.9. File Serializer
- 1.2.10. Enhanced Error Reporting
- 1.2.11. XPL Profiling Support
- 1.2.12. Other Changes
- 1.2.13. Other Incompatible changes

1.3. Changes in Version 2.8

- 1.3.1. XForms
- 1.3.2. Move to ObjectWeb and Product Name
- 1.3.3. Performance Improvements
- 1.3.4. Miscellaneous Bug Fixes

1.4. Changes in Version 2.7

- 1.4.1. JBoss Support
- 1.4.2. XForms
- 1.4.3. XPath 2.0 Support
- 1.4.4. User-Defined Processor Inputs
- 1.4.5. eXist Native XML Database
- 1.4.6. Examples Setup
- 1.4.7. Saxon Upgraded to Version 8.1.1
- 1.4.8. Documentation Printing
- 1.4.9. Processors
- 1.4.10. Command Line Interface

1.5. Changes in Version 2.6

- 1.5.1. XForms
- 1.5.2. Standard Representation of Binary and Text Documents
- 1.5.3. Improved Request Body and Uploaded Files Support
- 1.5.4. URL Generator
- 1.5.5. Serializers
- 1.5.6. Examples
- 1.5.7. Page Flow
- 1.5.8. Email Processor
- 1.5.9. Custom Processors
- 1.5.10. WebDAV Resource Manager
- 1.5.11. J2SE 5.0 Support

1.6. Installing Orbeon PresentationServer

- 1.6.1. Downloading
- 1.6.2. System Requirements
- 1.6.3. Installing OPS on Apache Tomcat
- 1.6.4. Installing OPS on BEA WebLogic 9.1
- 1.6.5. Installing OPS on BEA WebLogic 7.0 and 8.1
- 1.6.6. Installing OPS on IBM WebSphere 6
- 1.6.7. Installing OPS on JBoss 3.2.7 and 4.0
- 1.6.8. Security

2. Core Technologies Reference

2.1. XForms Reference

- 2.1.1. Scope
- 2.1.2. Introduction to XForms
- 2.1.3. Getting Started With the OPS XForms Engine
- 2.1.4. Programming With XForms 1.0

- 2.1.5. Formatting
 - 2.1.6. XForms Instance Initialization
 - 2.1.7. Relative Paths
 - 2.1.8. XForms and Services
 - 2.1.9. Extensions
 - 2.1.10. State Handling
 - 2.1.11. JavaScript Integration
- 2.2. Page Flow Controller
 - 2.2.1. Introduction to the Page Flow Controller
 - 2.2.2. Basics
 - 2.2.3. XML Submission
 - 2.2.4. Navigating Between Pages
 - 2.2.5. Other Configuration Elements
 - 2.2.6. Error Handling
 - 2.2.7. Typical Combinations of Page Model and Page View
 - 2.2.8. Examples
- 2.3. Page Flow Epilogue
 - 2.3.1. Introduction to the Page Flow Epilogue
 - 2.3.2. Basics
 - 2.3.3. The Standard Epilogue: epilogue.xpl
 - 2.3.4. The Servlet Epilogue: epilogue-servlet.xpl
 - 2.3.5. The Portlet Epilogue: epilogue-portlet.xpl
 - 2.3.6. XForms Processing
 - 2.3.7. The Standard Theme
- 2.4. Introduction to the XML Pipeline Definition Language (XPL)
 - 2.4.1. Introduction
 - 2.4.2. XPL Interpreter
 - 2.4.3. Namespace
 - 2.4.4. <p:config> element
 - 2.4.5. <p:param> element
 - 2.4.6. <p:processor> element
 - 2.4.7. <p:choose> element
 - 2.4.8. <p:for-each> element
 - 2.4.9. href attribute
 - 2.4.10. Processor Inputs and Outputs
- 3. OPS Technologies Reference
 - 3.1. Resource Managers
 - 3.1.1. Introduction
 - 3.1.2. General Configuration
 - 3.1.3. Filesystem Resource Manager
 - 3.1.4. ClassLoader Resource Manager
 - 3.1.5. WebApp Resource Manager
 - 3.1.6. URL Resource Manager
 - 3.1.7. WebDAV Resource Manager
 - 3.1.8. Priority Resource Manager
 - 3.1.9. Caché Database Resource Manager
 - 3.2. Properties File
 - 3.2.1. Overview
 - 3.2.2. Global Properties
 - 3.2.3. Java Processor Properties
 - 3.2.4. Email Processor Properties
 - 3.3. Non-XML Documents in XPL
 - 3.3.1. Introduction
 - 3.3.2. Binary Documents
 - 3.3.3. Text Documents
 - 3.3.4. Streaming
 - 3.4. Error Processor
 - 3.4.1. Rationale
 - 3.4.2. Configuration
 - 3.4.3. Example
 - 3.5. Listeners
 - 3.5.1. Servlet Context Listener
 - 3.5.2. Session Listener
 - 3.6. URL Rewriting
 - 3.6.1. Rationale

- 3.6.2. [Default URL Rewriting](#)
 - 3.6.3. [Working with URL Rewriting](#)
 - 3.6.4. [Known Limitations](#)
- 3.7. [XML Namespaces](#)
 - 3.7.1. [Introduction](#)
 - 3.7.2. [Namespace Usage in OPS](#)
- 4. [Processors Reference](#)
 - 4.1. [URL Generator](#)
 - 4.1.1. [Introduction](#)
 - 4.1.2. [Content Type](#)
 - 4.1.3. [XML Mode](#)
 - 4.1.4. [HTML Mode](#)
 - 4.1.5. [Text Mode](#)
 - 4.1.6. [Binary Mode](#)
 - 4.1.7. [Character Encoding](#)
 - 4.1.8. [HTTP Headers](#)
 - 4.1.9. [Cache Control](#)
 - 4.1.10. [Relative URLs](#)
 - 4.2. [Request Generator](#)
 - 4.2.1. [Introduction](#)
 - 4.2.2. [Configuration](#)
 - 4.2.3. [Request Body](#)
 - 4.2.4. [Uploaded Files](#)
 - 4.3. [Other Generators](#)
 - 4.3.1. [Introduction](#)
 - 4.3.2. [Scope Generator](#)
 - 4.3.3. [Servlet Include Generator](#)
 - 4.3.4. [Exception Generator](#)
 - 4.3.5. [Bean Generator](#)
 - 4.4. [HTTP Serializer](#)
 - 4.4.1. [Scope](#)
 - 4.4.2. [Configuration](#)
 - 4.4.3. [Content Type](#)
 - 4.4.4. [Binary Mode](#)
 - 4.4.5. [Text Mode](#)
 - 4.5. [Other Serializers](#)
 - 4.5.1. [Scope](#)
 - 4.5.2. [URL Serializer](#)
 - 4.5.3. [File Serializer](#)
 - 4.5.4. [Scope Serializer](#)
 - 4.5.5. [Null Serializer](#)
 - 4.5.6. [Flushing the Output Stream](#)
 - 4.5.7. [Legacy HTTP Serializers](#)
 - 4.6. [Converters](#)
 - 4.6.1. [Introduction](#)
 - 4.6.2. [Standard Converters](#)
 - 4.6.3. [To-XML Converter](#)
 - 4.6.4. [XSL-FO Converter](#)
 - 4.6.5. [XLS Converters](#)
 - 4.7. [XSLT and JAXP Processors](#)
 - 4.7.1. [Introduction](#)
 - 4.7.2. [Inputs and Outputs](#)
 - 4.7.3. [Processor Names](#)
 - 4.7.4. [Usage](#)
 - 4.7.5. [Passing Parameters to XSLT Stylesheets](#)
 - 4.7.6. [Streaming Transformations for XML \(STX\)](#)
 - 4.8. [XInclude Processor](#)
 - 4.8.1. [Introduction](#)
 - 4.8.2. [Note About XInclude in Orbeon PresentationServer](#)
 - 4.8.3. [Inputs and Outputs](#)
 - 4.8.4. [Usage](#)
 - 4.9. [XUpdate Processor](#)
 - 4.9.1. [Scope](#)
 - 4.9.2. [About XUpdate](#)

- 4.9.3. Language Reference
 - 4.9.4. Using the XUpdate Processor
 - 4.9.5. Examples
- 4.10. SQL Processor
 - 4.10.1. Introduction
 - 4.10.2. Inputs and Outputs
 - 4.10.3. Configuration Template
 - 4.10.4. Transactions Management
- 4.11. XML Databases
 - 4.11.1. Introduction
 - 4.11.2. Tamino XML Server 4.1
 - 4.11.3. XML:DB Databases (eXist)
- 4.12. LDAP Processor
 - 4.12.1. Introduction
 - 4.12.2. Usage
 - 4.12.3. Example
- 4.13. Directory Scanner
 - 4.13.1. Introduction
 - 4.13.2. Inputs and Outputs
 - 4.13.3. Configuration
 - 4.13.4. Output Format
 - 4.13.5. Ant Patterns
- 4.14. Delegation Processor
 - 4.14.1. Introduction
 - 4.14.2. Inputs and Outputs
 - 4.14.3. Calling a JavaBean
 - 4.14.4. Calling an EJB
 - 4.14.5. Calling a Web Service
- 4.15. Java Processor
 - 4.15.1. Rationale
 - 4.15.2. Benefits and Drawbacks
 - 4.15.3. Usage
 - 4.15.4. Compilation
 - 4.15.5. Compilation Class Path
 - 4.15.6. Runtime Class Loading
 - 4.15.7. Limitations
 - 4.15.8. Properties
 - 4.15.9. Example
- 4.16. Image Server
 - 4.16.1. Configuration
 - 4.16.2. Image Input
 - 4.16.3. Drawing
- 4.17. Charts and Spreadsheets
 - 4.17.1. Chart Processor
 - 4.17.2. Excel Processors
- 4.18. PDF Extraction Processor
 - 4.18.1. Rationale
 - 4.18.2. Usage
- 4.19. Email Processor
 - 4.19.1. Scope
 - 4.19.2. Data Input
 - 4.19.3. Simple Messages
 - 4.19.4. Character Encoding
 - 4.19.5. Message Parts
 - 4.19.6. Inline and Out of Line Parts
 - 4.19.7. Properties
 - 4.19.8. Examples
- 4.20. Yahoo Instant Messaging Processor
 - 4.20.1. Scope
 - 4.20.2. Config Input
 - 4.20.3. Data Input
 - 4.20.4. Example
- 4.21. Pipeline Processor

- 4.21.1. Introduction
 - 4.21.2. Inputs and Outputs
 - 4.21.3. Example of Use
 - 4.21.4. Configuration
 - 4.21.5. Optional Inputs
 - 4.21.6. Optional Outputs
- 4.22. Validation Processor
 - 4.22.1. Rationale
 - 4.22.2. Usage
- 4.23. Scheduler Processor
 - 4.23.1. Scheduler
- 4.24. Other Processors
 - 4.24.1. Resource Server
 - 4.24.2. Identity Processor
 - 4.24.3. Debug Processor
 - 4.24.4. Redirect Processor
- 5. API Reference
 - 5.1. Processor API
 - 5.1.1. Scope
 - 5.1.2. Why Write Custom Processors?
 - 5.1.3. Prerequisites
 - 5.1.4. Processors With Outputs
 - 5.1.5. Processors With No Output
 - 5.1.6. Processor State
 - 5.1.7. Using custom processors from XPL
 - 5.2. Pipeline Engine API
 - 5.2.1. Introduction
 - 5.2.2. API
- 6. Integration
 - 6.1. Authentication
 - 6.1.1. Introduction
 - 6.1.2. Restricting Access using web.xml
 - 6.1.3. Mapping Roles to Users
 - 6.1.4. Accessing Security Information From the Application
 - 6.1.5. Logout
 - 6.2. Command Line Applications
 - 6.2.1. Introduction
 - 6.2.2. Command Line Interface
 - 6.2.3. Examples
 - 6.3. Web Services
 - 6.3.1. Overview
 - 6.3.2. Consuming Web Services
 - 6.3.3. Exposing Web Services
 - 6.4. Packaging and Deployment
 - 6.4.1. Scope
 - 6.4.2. WAR Structure
 - 6.4.3. OPS Initialization
 - 6.4.4. Main Processor
 - 6.4.5. Error Processor
 - 6.5. EJB Support
 - 6.5.1. Classification
 - 6.5.2. Calling Existing EJBs
 - 6.5.3. Processor EJB Encapsulation

1. Getting Started

1.1. Welcome

1.1.1. Introduction

Welcome to the reference documentation for the Orbeon PresentationServer (OPS) platform.

OPS is an open source J2EE-based platform for XML-centric web applications. OPS is built around XML technologies such as XHTML, XForms, XSLT, XML pipelines, and web services, which makes it ideal for applications that capture, process and present XML data.

Unlike other popular web application frameworks like Struts or WebWork which are based on Java objects and JSP, OPS is based on XML documents and XML technologies. This leads to an architecture better suited for the tasks of capturing, processing, and presenting information in XML format, and often does not require writing any Java code at all to implement the presentation layer of your web application.

OPS is built around Orbeon's [XPL engine](#), a mature, high-performance XML pipeline engine for processing XML data.

1.1.2. Purpose of the User Guide

The purpose of this guide is to get you acquainted with the main components of OPS:

- **XForms.** XForms is the W3C standard for creating complex web forms.
- **The Page Flow Controller (PFC).** The PFC is the heart of your OPS application. It defines your application's pages and how you navigate between them.
- **The XML Pipeline Definition Language (XPL).** XPL allows you to implement lightweight processes that handle XML data, without writing a single line of Java or other scripting language.
- **OPS's built-in XML components.** The built-in OPS XML components provide services such as accessing databases, performing data transformations, calling web services, and more. If you need more, you can write your own XML components.

1.1.3. Prerequisites

This guide should be accessible to anyone with some experience developing simple web applications, but it is helpful to understand the basics of XML, Extensible Stylesheet Language Transformations (XSLT), and XForms beforehand. For more information, consult the following sources as a starting point:

- [XML](#)
- [XSLT](#)
- [XForms](#)

In addition to online resources, many books about these technologies are available in printed form.

1.2. Changes in Version 3.0

1.2.1. Introduction

Welcome to Orbeon PresentationServer (OPS) 3.0!

OPS 3.0 features an XForms engine much improved over OPS 2.8's, significant improvements in the Page Flow Controller, and much more!

This document describes changes made between OPS 2.8 and OPS 3.0.

1.2.2. Known Limitations of OPS 3.0

The following limitations are known:

- XForms upload controls only work within forms that are submitted with a reference to a submission element with `replace="all"`. The value of XForms upload controls is ignored when forms are submitted with `replace="instance"` or `replace="none"`.
- At most one XML schema is allowed on a particular XForms model.
- Calculation dependencies are not computed as per the XForms 1.0 specification. With OPS 3.0, first the calculations are evaluated in the order they appear in your XForms model. Then the required, relevant, and readonly model item properties are evaluated, also in the order they appear in your XForms model. This means that you have to be careful with the order in which you declare your `<xforms:bind>` elements.
- `xforms:message` only supports `level="modal"`.
- The Ajax-based XForms engine currently works with Mozilla Firefox and Microsoft Internet Explorer 6. Support for Safari and Opera is planned.
- The Orbeon PresentationServer Tutorial has not been updated to reflect the new features and best practices of OPS 3.0.
- Initial generation of an XForms page may still have poor performance in some circumstances.
- Not all the OPS examples have been updated to reflect the new XForms engine capabilities.
- The OPS reference XForms documentation is a work in progress. Please refer to the "XForms NG" examples for details.

- The OPS reference XForms compliance matrix has not yet been updated to reflect all the changes in OPS 3.0.
- Migration documentation from 2.8 to 3.0 is not complete yet.
- The OPS Blog example is not complete.
- The PDF version of the User Guide does not have page numbers.

Please also visit the [OPS bug tracker](#) on the ObjectWeb Forge, and report and discuss issues in the [ops-users mailing-list](#).

1.2.3. XForms

The OPS XForms engine introduces a big step towards supporting all of the XForms 1.0 specification. The main changes are described below. Please also visit the new XForms examples, in particular the [XForms Controls](#), the [BizDoc NG](#) example, and the [XForms Sandbox](#) example. XForms improvements in OPS 3.0 include but are not limited to:



- **Ajax-based engine.** The XForms engine is now based on Ajax technologies. This makes the XForms engine much more responsive to user interaction than with OPS 2.8.

Note

We sometimes informally refer to the XForms engine present in OPS 2.8 and earlier (but also present in OPS 3.0 for backward compatibility) as the *Classic XForms engine* or, in short, as *XForms Classic*. We refer to the new Ajax-based XForms engine, present in OPS 3.0 and forward, as the *Next Generation XForms engine* or, in short, as *XForms NG*.

- **Standard XHTML integration.** The XForms engine works on standard XHTML + XForms documents, without the need for a separately described XForms model as was the case with OPS 2.8. XForms models are simply included in the XHTML page view under the `<xhtml:head>` element, as recommended by XForms 1.0. In this mode, you don't use the `xforms` attribute on the PFC `<page>` elements. The `xforms` attribute is still supported for backward compatibility, but its use triggers the use of the legacy OPS 2.8 XForms engine.
 - **XForms event model.** The XForms engine supports the XForms event model, including most XForms events and XForms actions.
 - **Multiple XForms models and instances.** The XForms engine supports multiple models and multiple instances within models. The `instance()` function is supported.
 - **XForms switch module.** The XForms engine supports the XForms switch module.
 - **XForms repeat module.** The XForms engine supports correctly the current index in repeated sections, including the `index()` function.
 - **XForms submission.** The XForms engine supports submitting forms as `application/xml` with HTTP POST and `application/x-www-form-urlencoded` with HTTP GET (including to external applications). Support for `replace="instance"`, `replace="all"`, and `replace="none"` is included.
 - **XForms Range control.** The XForms engine supports a subset of the functionality of the XForms Range (or slider) control.
 - **Extension functions.** The XForms engine supports the `xxforms:call-xpl()` extension function to call arbitrary XPL programs from XForms, as well as some eXforms functions.
 - **Dynamic XForms models.** The new XForms engine and PFC improvements now allow to easily generate XForms models based on XML submissions. With OPS 2.8, this was much more difficult as this had to be done in a dynamic XForms model pipeline, which did not have access to an XML submission.
 - **Simplified theme.** The default theme and XForms make more use of CSS so that configuration of your own theme is easier.
 - **XForms sandbox.** The new [XForms Sandbox](#) example allows you to easily try your own XForms examples. Just write an XHTML + XForms example, upload it, and watch the results!
 - **Other changes.** The XForms engine supports the `value` attribute on `xforms:output`, handles inheritance of model item properties, and includes numerous improvements and bug fixes.
- XSLT views now must always use `doc('input:instance')` to access a submitted XML instance instead of expecting the instance on their main input.
 - With OPS 2.8 if your XForms instance was:

```
<credi t-card>
  <type>vi sa</type>
  ...
</credi t-card>
```

and your first XForms element under the `<xhtml:body>` element was:

```
<xforms:group ref="credi t-card" xml ns:xforms="http://www.w3.org/2002/xforms"/>
```

then `credit-card` would evaluate against the *document node* of the unique XForms instance. The above code worked, but this behavior was incorrect. The XPath evaluation context for top-level XForms controls has to be the root *element* of the default XForms instance instead. This means that the code above must be changed in one of two ways:

```
<xforms:group ref="/credi t-card" xml ns:xforms="http://www.w3.org/2002/xforms"/>
```


or:

```
<xforms:group ref=". " xmlns:xforms="http://www.w3.org/2002/xforms" />
```

If you use absolute XPath expressions for your top-level XForms controls, no change is necessary.

- With OPS 2.8, the "required" model item property did not apply at all to elements validated with XML schemas. If for example a value was of type `xs:string` but missing and required, it was marked as valid (which was correct) but submission would pass. With OPS 3.0, nodes which are "valid", "required" and empty will cause submission to fail. Be sure to check your types and "required" model item properties.
- The `xxforms:choose`, `xxforms:when` and `xxforms:otherwise` constructs are no longer supported with the new XForms engine. Instead, use `relevance` and `xforms:group` or `xforms:switch` / `xforms:case`.

Warning

Your applications written against OPS 2.8 and run within OPS 3.0 without modifications uses the Classic XForms engine of OPS 3.0. In order to trigger the use of XForms NG and benefit from all the new XForms features of OPS 3.0, some code migration work is required. Note that it is possible to determine on a page by page basis whether XForms Classic or XForms NG is used, and therefore to progressively upgrade your application to XForms NG.

When migrating an application from XForms Classic to XForms NG, a series of tasks need to be performed, as the OPS XForms engine is now much closer from the XForms 1.0 specification. The following list attempts to describe the most common aspects of the migration. It does not intend to be inclusive:

- **XForms model migration.** For a given `<page>` element: if your OPS 2.8 XForms model is static (which is usually the case), copy it under your XHTML page view's `xhtml:head` element. Then remove the `xforms` attribute from the `<page>` element. You can then remove the file containing your XForms model.

```
<xhtml:head>
  <xhtml:title>My Title</xhtml:title>
  <xforms:model id="my-model">
    <xforms:instance id="main">
      ...
    </xforms:instance>
    ...
  </xforms:model>
</xhtml:head>
```

Another possibility is to keep your XForms model file, and to XInclude that file in your XHTML page view under the `xhtml:head` element.

```
<xhtml:head>
  <xhtml:title>My Title</xhtml:title>
  <xi:include href="my-model.xml"/>
</xhtml:head>
```

Here again, you must remove the `xforms` attribute from the `<page>` element.

If your XForms model is dynamic, then you can use XSLT in your page view to dynamically produce the model:

```
<xhtml:head>
  <xhtml:title>My Title</xhtml:title>
  <xforms:model id="my-model">
    <xforms:instance id="main">
      <form>
        <name>
          <xsl:value-of select="doc('input:data')/*/my-name"/>
        </name>
        ...
      </form>
    </xforms:instance>
    ...
  </xforms:model>
</xhtml:head>
```

The PFC now provides XML submissions on the page models, views and actions' `instance` inputs. This allows for easily building dynamic XForms models from an XML submission, including an XForms submission, which was difficult to do before. In the example above, the current XML submission can be accessed from XSLT in the page view with `doc('input:instance')`.

For more information, visit the XForms Reference's [XForms Instance Initialization](#) section and the Page Flow Controller's [XML Submission](#) section.

- **Required attributes.** Some controls now require certain attributes, as per the XForms 1.0 specification. In particular:
 - `xforms:submit` now requires a `submission` attribute.
 - XForms actions, including `xforms:setvalue`, need an `ev:event` attribute, or need to be enclosed within an `xforms:action` element with an `ev:event` attribute.
 - The `xforms:submission` element now requires an `id` attribute to be useful (by being referred from the `submission` attribute of `xforms:submit` or `xforms:send`). Attributes `action` and `method` are also mandatory.

- **Validation processing model.** As per the XForms specification, invalid XForms instances, or XForms instances with missing required elements cannot be submitted and instead throw `xforms-submit-error` events. This means that there is no longer a need to check on `xxforms:valid` attributes in the page flow to determine whether a submitted XForms instance is valid or not.
- **Page flow roundtrips.** `xforms:repeat` updates (threw actions such as `xforms:insert` and `xforms:delete`) are no longer visible in the page flow. With XForms Classic, upon inserting or deleting a row with those actions, the entire form would be submitted, giving an opportunity to the page flow to intercept and regenerate the page. With XForms NG, these actions operate within the XForms engine.

Use XForms submissions with `replace="none"` or `replace="instance"` if you need to submit an XForms instance to the page flow before or after performing such actions.
- **XForms events and actions.** Many operations can now be performed with XForms events and actions, rather than using `xforms:setvalue` in an `xforms:submit` and then performing a complete submission.
- **XML services and page flow.** Because the XForms engine now operates completely separately from the page flow, XForms pages must now explicitly perform XForms submissions to interact with pages defined in your page flow.

In particular, when using an XForms submission with `replace="none"` or `replace="instance"`, the page flow can now be used to implement *XML services*, that is services that receive XML (through the XForms submission) and produce XML as a result. Such services are implemented in your page flow as usual with `<page>` elements, and differ with regular pages only in that they either do not return any result, or return an XML document as result as opposed to return a complete HTML page.

1.2.4. Page Flow

Several improvements have been added to the Page Flow Controller, some of them motivated by the new XForms engine. The PFC is now more generic and less tied to the built-in OPS XForms implementation. At the same time it plays better than before with XForms, including client-side XForms engines. The major PFC concepts found with OPS 2.8 have not changed and backward compatibility is kept. The main changes are the following:

- **Documentation.** The [PFC documentation](#) has been reworked and greatly improved.
- **Standard Epilogue.** The standard epilogue has been restructured into three separate files so as to be easier to understand, modify, and extend. It is [fully documented](#).
- **Deprecation of the `xforms` attribute.** Using XForms with OPS no longer implies using an `xforms` attribute on the `<page>` element. Instead, XForms models are included in the XHTML page view under the `<xhtml:head>` element, as recommended by XForms 1.0. The `xforms` attribute is still supported for backward compatibility, and triggers the use of the legacy OPS 2.8 XForms engine.
- **New XML submission mechanism.** The PFC features a new generic XML submission mechanism. Each page in the PFC, instead of supporting native OPS XForms engine submissions, now supports generic XML submissions. You submit XML by POST-ing content with an XML content-type (other types of XML submissions can be added). XML submissions can be performed from external applications (through Web Services, XML-RPC, etc.), client-side XForms engines, the built-in OPS XForms engine, or by the PFC itself when navigating between pages.
- **Deprecation of the `<param>` element.** The PFC's `<param>` element was used to set values into a submitted XForms instance. A new, more flexible element, `<setvalue>`, now performs the same task. The `<setvalue>` element supports extracting information from regular expressions applied to the request path, as well as extracting request parameters. This allows for creating "clean", REST-like URLs in your application. `param` is still supported for backward compatibility.
- **Generic XML submission transformations.** The PFC features a new generic and extensible XML submission transformation mechanism. With OPS 2.8, XUpdate code had to be used to transform XML instances between pages. XUpdate support in the PFC is now deprecated. Instead, XSLT or XQuery should be used for that purpose. For more information, please refer to the [<result> element section](#) of the PFC documentation.
- **Accessing XML submissions.** XML submissions can be accessed from actions, page models, and page views, through the `instance` input. Users should not assume that XML submission are available from the `data` inputs.
- **Shorter page flows.** Because of the enhanced XForms support, in particular, the XForms switch module and support for `application/xml` submissions, page flows are typically shorter to write. Compare for example the [BizDoc Classic](#) example with the [BizDoc NG](#) example.

The following incompatible changes have been made:

- **Optional Epilogue Output.** Support for the legacy `data` output of the epilogue has been dropped. With OPS 2.8 and earlier, the epilogue could have a `data` output. The PFC was then in charge of HTML serialization. This is no longer possible: serialization must occur in the epilogue itself. If you have code relying on this feature, simply remove the epilogue's `data` output and add an HTML serializer to your epilogue.

1.2.5. XSLT processor

The following improvements have been made to the XSLT processor:

- **Attributes input.** The [XSLT processor](#) supports a new `attributes` input, which allows setting JAXP TransformerFactory attributes.
- **Safer defaults.** The default [XSLT processor](#) (accessed with `oxf:xslt`, and configured in `processors.xml`) no longer allows executing external functions. To enable external functions, use `oxf:unsafe-xslt`, or configure the XSLT processor's `attributes` input. This makes the XSLT processor safer by default.
- **Default implementation.** The default XSLT 1.0 processor implementation, configured in `processors.xml`, is now Saxon 8 instead of Xalan. The default XSLT 2.0 implementation remains Saxon 8, as was the case before.

- **Deprecated behavior.** The `XSLT processor` used to support a value of `DEFAULT` or `interpreter` for the transformer input, for backward compatibility. This is no longer supported. The transformer must provide only a valid JAXP TransformerFactory class name. Most users should not be affected by this change.

Note

These changes should not affect applications that used `oxf:xslt` unless stylesheets use external functions. In this case, you have to use `oxf:unsafe-xslt` instead.

If you were using `oxf:xslt-1.0` or were using XSLT 1.0 stylesheets directly referenced from a page flow, and used Xalan-specific features, you have to either convert your stylesheet to use Saxon features instead, or use `oxf:xalan`.

1.2.6. XQuery processor

The XQuery processor has been updated:

- **Default implementation.** The default implementation of the XQuery processor is now Saxon 8.
- **Safer defaults.** This processor is available with `oxf:xquery` or `oxf:unsafe-xquery`. The former does not allow calling external functions by default, while the latter does, similar to the new behavior of the XSLT processor.
- **Query format.** In addition to XQuery code embedded into XML, the XQuery processor now supports in its `config` input a text document of the form:

```
<document xsi:type="xs:string">
  xquery version "1.0"; ...
</document>
```

- **PFC integration.** XQuery can be used in the PFC to perform XML submission transformations.

1.2.7. XHTML Support

XHTML is now much better supported:

- **Simplified theme.** There is no need to remove the XHTML namespace from XSLT, as was done before in `theme.xsl`. Your theme should simply use the XHTML namespace.
- **Switching between XHTML and HTML.** The standard epilogue illustrates how to generate XHTML and HTML to different browsers.
- **Using XHTML serialization.** The standard epilogue illustrates how to connect the XHTML serializer.

Note

In order to benefit from features such as XForms, you should make sure that your page views generate XHTML. In particular, your XHTML elements must in the XHTML namespace (<http://www.w3.org/1999/xhtml>).

1.2.8. SQL Processor

The following changes have been made to the SQL processor.

- **Stored Procedures.** The SQL processor now uses the JDBC `CallableStatement` interface when `sql:call` is used instead of `sql:query`. This allows for calling stored procedures using the JDBC escape syntax, for example:

```
<sql:call xmlns:sql="http://orbeon.org/oxf/xml/sql">
  { call SalesByCategory(<sql:param type="xs:string" select="*/category"/>, <sql:param type="xs:int"
    select="*/year"/>) }
</sql:call>
```

Note

OUT and INOUT parameters are not yet supported.

- **Multiple Result-Sets.** The SQL processor now supports multiple result-sets. It is possible to handle the result-sets returned by a query or call individually for each result-set, or globally for all result-sets, using the `sql:result-set` element. The optional `result-sets` attribute specifies how many result-sets are handled by a given `sql:result-set` element. If not specified, the default is one result-set. If the value is unbounded, the `sql:result-set` element handles all the remaining result-sets returned by the statement execution. Otherwise, a positive number of result-sets must be specified.

```
<!-- Handle the first two result-sets -->
<sql:result-set result-sets="2" xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <my-first-result-sets>
    <sql:row-iterator>
      <row>
```

```

        <sql: get-col umns format="xml" />
    </row>
</sql: row-i terator>
</my-fi rst-resul t-sets>
</sql: resul t-set>
<!-- Handle All the remaining result-sets -->
<sql: resul t-set resul t-sets="unbounded" xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
    <my-other-resul t-sets>
        <sql: row-i terator>
            <row>
                <sql: get-col umns format="xml" />
            </row>
        </sql: row-i terator>
    </my-other-resul t-sets>
</sql: resul t-set>
<!-- This will be executed if no row was returned by any result-set -->
<sql: no-resul ts xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
    <there-are-no-resul ts/>
</sql: no-resul ts>

```

sql:no-results has been updated to execute when none of the previous sql:result-set elements returned rows.

- **Column Iterator.** The SQL processor is now able to explicitly iterate over all the columns returned by a result-set with the sql:column-iterator element. A column iterator can be used under the sql:result-set element, or under the sql:row-iterator element. This allows for example easily extracting column metadata:

```

<sql: resul t-set xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
    <metadata>
        <sql: col umn-i terator>
            <col umn>
                <sql: attri bute name="i ndex">
                    <sql: get-col umn-i ndex/>
                </sql: attri bute>
                <sql: attri bute name="name">
                    <sql: get-col umn-name/>
                </sql: attri bute>
                <sql: attri bute name="type">
                    <sql: get-col umn-type/>
                </sql: attri bute>
            </col umn>
        </sql: col umn-i terator>
    </metadata>
</sql: resul t-set>

```

- **Result-Set Metadata.** The SQL processor is now able to retrieve result-set metadata, with the following new elements. The must be used within a sql:column-iterator element, unless a column-name or column-index attribute is explicitly specified:

- sql:get-column-index: retrieves the current column index.
- sql:get-column-name: retrieves the current column name.
- sql:get-column-type: retrieves the current column type name as returned by result-set metadata.

- **Outputting Attributes.** The SQL processor is now able to dynamically generate new attributes with the sql:attribute element, for example:

```

<sql: attri bute name="i ndex" xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
    <sql: get-col umn-i ndex/>
</sql: attri bute>

```

- **Optional XML Type Information.** The XML type (specified with type="xs:int", for example) is now optional on column getters. If not specified, a default type obtained from the result-set metadata is used to determine how to best get the value from the result-set.
- **New Element and Attribute Names.** Some element and attributes have been renamed. The old names are still supported for backward compatibility:
 - sql:results has been renamed to sql:result-set.
 - sql:row-results has been renamed to sql:row-iterator.
 - sql:get-column has been renamed to sql:get-column-value, for consistency with the new sql:get-column-* elements.
 - column-name is now the standard attribute name for identifying a column name on the sql:get-column-*, sql:get-column, and sql:group elements.

The following incompatible changes have been made:

- **Legacy Types.** Backward compatibility with pre-OPS 2.0 int and string types has been removed. Use xs:int and xs:string instead, while declaring the xs prefix as explained below.

- **Implicit Type Prefixes.** Legacy support for implicit type prefixes (`xs` and `oxf`) is deprecated. Backward compability is enabled with the `legacy-implicit-prefixes` property as follows:

```
<property as="xs:boolean" processor-name="oxf:sql" name="legacy-implicit-prefixes" value="true"/>
```

When this property is missing or set to `false`, type prefixes must be mapped as is customary for XML vocabularies. Add the following namespace declarations: `xmlns:xs="http://www.w3.org/2001/XMLSchema"` and `xmlns:odt="http://orbeon.org/oxf/xml/datatypes"`. Doing so then allows using data types as before, for example `xs:string` or `odt:xmlFragment`.

- **Stricter Type Checking.** Column getters are now checked against result-set column metadata. In the past, using `xs:string` for a column of type `INTEGER` was allowed. Now if an XML type is specified, the XML type must match the SQL type. Alternatively, as mentioned above, it is possible to not specify an XML type at all.

1.2.9. File Serializer

The File serializer now works like the HTTP serializer: it only accepts [text](#) and [binary](#) documents as input, and no longer handles conversions to XML, HTML, XHTML or text. To serialize to a file XML, HTML, XHTML or plain text, connect the [XML](#), [HTML](#) or [Text converters](#) to the File serializer.

The legacy, deprecated File serializer is still available as `oxf:legacy-file-serializer`.

1.2.10. Enhanced Error Reporting

The following enhancements have been made to the error reporting on OPS:

- **OPS Stack Trace.** When a Java exception occurs, OPS now reports the OPS stack trace, which provides useful OPS-oriented information about the events leading to an exception.
- **Java Stack Traces.**
 - Java stack traces are split into their different request components. For example, an exception may have been produced by a request on the `OPSServlet`, forwarded again to `OPSServlet`, then going through the OPS example portal's `OPSPortlet`. In this case, the stack trace is split into three parts for clarity.
 - Some coloring is applied to the class names to easily distinguish OPS classes from third-party classes.
 - As OPS users may have noticed, Java stack traces can be quite long due to the streamed nature of the execution of XPL pipelines. Exceptions are now initially folded, except for the first few lines, and expandable on click.
- **Separate Stylesheet.** The XSLT stylesheet used to format stack traces is not in a separate file: `oxf:/config/error.xsl`.

Like with OPS 2.8 and earlier, the layout of the stack traces can be customized: simply edit `oxf:/config/error.xpl` and `oxf:/config/error.xsl`. More advanced configuration can be performed by changing the [error processor](#).

1.2.11. XPL Profiling Support

- A trace API has been added. The interface is `org.orbeon.oxf.pipeline.api.PipelineContext.Trace`
- OPS ships with two implementations of `Trace`, `org.orbeon.oxf.processor.NetworkTrace` and `org.orbeon.oxf.processor.StdoutTrace`. `NetworkTrace` sends profiling information to Studio which then displays the data in the trace views. `StdoutTrace` simply dumps profiling information to standard out.
- [Properties](#) for configuring the tracing have been added.
- Views for viewing profiling data have been added to Orbeon Studio.

1.2.12. Other Changes

The following changes are new in OPS 3.0:

- **HTML Documentation.** The OPS User Guide in static HTML format, broken in OPS 2.8, is now working correctly. This format allows you to consult the documentation offline without running OPS or without accessing the online documentation.
- **PDF Documentation.** The OPS User Guide in PDF version is back! You can print this book for reference.
- **OPS Blog example.** This [new example](#) illustrates many of the capabilities of OPS, including:
 - Implementing XML-RPC services
 - Connecting to a native XML database
 - Using XForms
 - Producing XHTML, RSS and other formats from a single data source
 - Implementing configurable themes with XSLT
 - Creating "clean" URLs in a REST perspective

- **Resource Manager.** The Flat File and [Filesystem](#) resource managers have been merged. The Flat File resource manager is now deprecated, and the Filesystem resource manager should be used instead. The Flat File resource manager can still be used for backward compatibility. The main difference between the two resource managers is that the Filesystem resource manager can be configured with or without a sandbox.
- **Directory Scanner Processor.** The purpose of the [Directory Scanner](#) processor is to analyse a directory structure in a filesystem and to produce an XML document containing metadata about the files, such as name and size. It is possible to specify which files and directories to include and exclude in the scanning process. The Directory Scanner is also able to optionally retrieve image metadata. Two new command-line examples illustrate the use of this processor.
- **Faster URL Rewriting.** In order to improve performance URL rewriting has been re-implemented in Java. The new implementation is accessed through the new rewrite processors `oxf:html-rewrite` and `oxf:html-rewrite`.
- **Disabled Validation.** Previously the data input of the `oxf:xforms-output` processor and the config input of the `oxf:portlet-include` processor were validated unless one turned off all input parameter validation in OPS. However we found that validation of these two inputs came with a significant price and consequently these inputs are no longer validated by default. To enable the validation one must specify the appropriate schema URI with the `schema-uri` attribute. For the data input of `oxf:xforms-output` the URI is `http://www.w3.org/2002/xforms/controls` and for the config input of `oxf:portlet-include` the URI is `http://orbeon.org/oxf/xml/portlet-include`.
- **URL Generator.** The [URL Generator](#) now has an option to specify whether XInclude processing must occur at XML parsing time.
- **JAR and WAR Names.** `orbeon.jar` has been renamed to `ops.jar`, `cli-orbeon.jar` has been renamed to `cli-ops.jar` and `orbeon.war` has been renamed to `ops.war`.
- **Delegation Processor.** The `select` attribute on the `operation` element is now documented. The default behavior when `select` is not present has changed when calling document-style web services. In that case, now the content of the SOAP body element is returned, instead of the content of the first element contained by the SOAP body.
- **New To-XML Converter.** The new [To-XML Converter](#) allows producing parsed XML documents from a binary document format.
- **New XInclude Processor.** The new [XInclude processor](#) allows processing XInclude instructions. XInclude was already supported at the parser level (Xerces support) before OPS 3.0.
- **Email processor.** The [Email processor](#) now supports setting custom email headers with the new `header` configuration element.
- **Listeners.** The [Listeners](#) documentation has been updated to reflect the correct configuration parameters. Listeners now produce better logs.
- **Servlet and Portlet Classes.** Servlet and Portlet classes have been renamed as follows:
 - `OPSServlet` replaces `OXFServlet`
 - `OPSPortlet` replaces `OXFPortlet`
 - `OPSServletFilter` replaces `OXFServletFilter`
 - `OPSServletContextListener` replaces `OXFServletContextListener`
 - `OPSSessionListener` replaces `OXFSessionListener`

The old names still work for backward compatibility.

1.2.13. Other Incompatible changes

The following incompatible changes are new in OPS 3.0:

- **Request Generator.** The [Request Generator](#) used to return a constructed path for the `path-info` parameter. It now correctly returns the Servlet API's path info. To get a complete path, use `request-path` instead.

1.3. Changes in Version 2.8

1.3.1. XForms

- It is now possible to have multiple `<xforms:alert>` in an XForms control. They will be all kept by the XForms engine and transformed into `<f:alert>`. The theme can then decide which one should be displayed. The default theme displays all the alerts.

You can use multiple `<xforms:alert>` for internationalization by adding on each `<xforms:alert>` an `xml:lang` attribute (e.g. `xml:lang="en"`, `xml:lang="fr"`) and selecting in your theme the appropriate message to display for the current user based on his preferred language.
- XForms schema validation can now be used to [validate attributes in the XForms instance](#). Note that the `xxforms:valid="false"` attribute is still only added on invalid elements in the XForms instance. The XForms engine does not annotate the XForms instance for invalid attributes.
- Instance nodes of type `xs:date` and `xs:time` bound to an XForms input control are rendered with an [appropriate date or time widget](#). You set the type of an instance node with a `<xforms:bind nodeset="/path/to/your/node" type="xs:date"/>` in your XForms model.
- As the XForms engine validates the XForms instance with a schema and evaluates the model item properties on the instance, it annotates the instance with attributes using the `xxforms` prefix. For instance, if an element is validated, an attribute `xxforms:valid="true|false"` is added on the element. In previous versions of OPS you could override the value set by the XForms engine in the MVC model or MVC view by modifying the value of those attributes in the instance. As the instance can be modified in the MVC model and view, the model item properties are reevaluated and the instance revalidated with the schema after the MVC view, in the epilogue. At that point, for example, if there is a `xxforms:valid="false"` attribute on a given element, but validation with the schema and model item properties returns "true", the XForms engine does not know if the content of the element was modified in the MVC model or view and the error corrected (in which case the element should be considered valid), or if the `xxforms:valid="false"` was added by the MVC model or view to tell the XForms engine that this element is invalid (in which case the element should be considered invalid).

To solve this problem, starting with OPS 2.8, you cannot override the annotations added by the XForms engine on the instance. Those annotations should be considered read-only. Instead, to instruct the XForms engine that a given node is invalid, use the method described in the [Custom Validation](#) section of the XForms reference. The benefit of this method is that it does not require the use of proprietary XForms extensions and that it can be used on elements as well as attributes.

1.3.2. Move to ObjectWeb and Product Name

- Orbeon [joined the ObjectWeb consortium](#) and the project is now hosted on the [ObjectWeb Forge](#).
- The name of the product is now written Orbeon PresentationServer (in one word).

1.3.3. Performance Improvements

Memory usage has been improved, which leads to significant performance improvements under heavy load or when the JVM is running with a relatively small heap, as the JVM spends less time doing garbage collection. Our testing shows performance improvement that range from 5% to 20% over version 2.7.

1.3.4. Miscellaneous Bug Fixes

- [XUpdate removing attributes and namespace declarations](#) — When using the `<xu:xupdate>` instruction to update the content of an element, the existing attributes and namespaces of the element were incorrectly removed.
- [XForms `xhtml:onclick` attribute](#) — The attribute `xhtml:onclick` on XForms submit control was ignored.
- [Rendering issues of the documentation under Safari](#) — Those issues have been solved.
- Other bugs have been fixed relative to the URL generator, Request generator, HTTP serializer, and server-side forwards following a file upload.

1.4. Changes in Version 2.7

1.4.1. JBoss Support

JBoss is now supported. See the [installation instruction](#) for more information on how to setup OPS on JBoss.

1.4.2. XForms

In line with our commitment to fully support the XForms specification, we now clearly document the level of support for each part of the specification in a [compliance matrix](#).

The `<xforms:select>` and `<xforms:select1>` controls let the end-user select one or more items from a list. You can now add the `selection="open"` attribute on those controls; when specified, the end-user can enter a value of its choice in text field, or choose one or more of the suggested values.

You can now use the `src` attribute on `<xforms:label>`, `<xforms:hint>`, `<xforms:help>`, `<xforms:alert>`, and `<xforms:message>` to specify the URL from where the text should be loaded, instead of having the text inline in the XForms view or in the XForms instance (`ref` or `bind` attribute).

In your XForms views, in addition of using the `ref` and `nodeset` attributes to reference nodes of the XForms instance, you can now use the `bind` attribute. You first assign an id to a nodeset in the XForms model: e.g. `<xforms:bind nodeset="/company/ceo" id="ceo"/>`; you then reference this id in your XForms view: e.g. `<xforms:input bind="ceo"/>`.

You can now nest `<xforms:bind>` elements in the XForms model. The `nodeset` XPath expression in nested elements is relative to the nodeset selected by the `nodeset` expression on the parent element.

You can now use the XPath 2.0 `doc()` function in XPath expressions in the XForms model. For instance, assuming that you have a "lookup table" defined in an XML file mapping department id to department name, you can write a model item property calculating the value of `<department-name>` element based on the value entered by the user for `<department-id>`:

```
<xforms:bind nodeset="/instance/department-name" calculate="doc('oxf:/departments.xml')/departments/department
[@id = /instance/department-id]/name"/>
```

In this version the names generated by OPS for HTML form elements are much shorter than in previous versions. Under certain circumstances (low bandwidth between server and end-user, large forms, complex XForms instances with many levels of nested elements using namespaces), submitting a form was slow as a lot of data had to be sent from the browser back to the server. A new name generation scheme has been implemented in this version. It solves the problem and guarantees that names will be short even if your XForms instance has many levels of nested elements and is using namespaces.

In previous versions of OPS, when a form was submitted and the XForms instance recreated, all the elements and attribute where in the right namespace, but the actual prefix was not necessarily kept and namespace declarations were inserted only where used. In this version, Presentation Server keeps the exact value of the prefixes and the location of namespace declaration when a form is submitted and the XForms instance is recreated.

The XForms extension control `xxforms:hidden` is supported again and documented.

The `<xxforms:choose>` used to run all the "true" `<xxforms:when>`. Instead, it now only run the first one if any, and the `<xxforms:otherwise>` otherwise. This is inline with the behavior of the `<xsl:choose>`.

`<xforms:message>` is now supported. This lets you display a message in a dialog when a submit control is clicked. See the [message action](#) documentation for more details.

1.4.3. XPath 2.0 Support

Using XPath 2.0 expressions is now supported in:

- XPL: for XPath expressions, test expressions in a `<p:when>`, and select expressions in a `<p:for-each>`.
- Page Flow: for when expressions in `<action>` and `<result>`.
- XForms: binding expressions in the XForms view (`ref` and `nodeset` attribute), XPath expressions in model item properties (`<xforms:bind>`), and XPath expressions in XForms actions.

1.4.4. User-Defined Processor Inputs

User-defined processor inputs, used by several processors including the XSLT processor and the Email processor, are now documented ([XPL documentation](#), [XSLT documentation](#)). The URI schemes to access user-defined inputs and outputs are now `input:` and `output:`, to avoid confusion with the `oxf:` scheme used for the OPS resource manager sandbox.

1.4.5. eXist Native XML Database

OPS ships with the latest stable version of eXist (eXist-1.0b2-build-1107). This fixes the issue of connecting to an external eXist instance. Here is an example `datasource.xml` to connect to a standalone eXist:

```
<datasource>
  <!-- Specify the driver for the eXist database -->
  <driver-class-name>org.exist.xml.db.DatabaseImpl</driver-class-name>
  <!-- Example of URL for a non-embedded use -->
  <uri>xml db:exist://localhost:8080/exist/xmlrpc</uri>
</datasource>
```

1.4.6. Examples Setup

All the examples using the SQL processor now work out of the box without configuring datasources at the application server or servlet container level. This includes the following examples:

- Address Book
- Employees
- Data in multiple formats
- XForms Upload
- Email

1.4.7. Saxon Upgraded to Version 8.1.1

Saxon is the default XSLT engine and is used throughout OPS for XPath 2.0 expressions. Saxon 8.1.1 is now bundled with OPS. See the [changes section](#) of the Saxon documentation for more details.

1.4.8. Documentation Printing

Printing the documentation has been improved through the use of CSS. As you print a given section of the documentation the left sidebar showing the chapters will not be printed.

1.4.9. Processors

- The SQL processor now supports external datasource definitions using the optional `datasource` input. Those directly refer to connections without using JNDI names mapped by the container. This allows examples to work out of the box. This is an example of datasource definition:

```
<datasource>
  <!-- Specify the driver for the database -->
  <driver-class-name>org.hsqldb.jdbcDriver</driver-class-name>
  <!-- This causes the use of the embedded database -->
  <uri>jdbc:hsqldb:file:orbeondb</uri>
  <!-- Optional username and password -->
  <username>sa</username>
  <password/>
</datasource>
```

- OPS ships with a recent version of HSQLDB (version 1.7.2.8).
- The SQL processor now supports reading `BINARY`, `VARBINARY` and `LONGVARBINARY` columns in addition to `BLOB`.
- The SQL processor's `data` input and output are now optional. This removes the need for "dummy" input content or Null serializers with the SQL processor.

The [URL generator](#) now supports URLs relative to the location where the generator is used, instead of just absolute URLs.

The [URL generator](#) also deals correctly with XInclude dependencies. For example, if a file called `file1.xml` includes a file called `file2.xml`, which in turn includes a file called `file3.xml`, and you modify `file3.xml`, then `file1.xml` is re-read automatically. Of course, this works with other dependency scenarios as well!

The [Java processor](#) now works correctly in command line mode. It also supports relative source paths and supports specifying alternate compilers.

The [Email processor](#) now supports `Cc` and `Bcc` recipients. See the [Email example](#) for an illustration.

The [Scope generator](#) is now able to marshall regular JavaBeans in addition to XML stored as Strings, W3C and dom4j documents. This functionality deprecates the [Bean generator](#). These two processors previously had some overlap in functionality.

The error processor now follows the same configuration mechanism as the main processor. The following property names have been deprecated:

- `oxf.servlet.error.processor.uri`
- `oxf.servlet.error.input.*`

The new names are as follows:

- `oxf.error-processor.name`
- `oxf.error-processor.input.*`

Visit [Packaging and Deployment](#) for more information on configuring the error processor, as well as [Error Pipeline](#) for information about creating an error pipeline.

1.4.10. Command Line Interface

The command to invoke is now:

```
java -jar cli-orbeon.jar ...
```

instead of:

```
java -jar orbeon.jar ...
```

1.5. Changes in Version 2.6

1.5.1. XForms

The XForms Upload control has been fixed. It comes with an [example](#) and is documented in the [XForms reference documentation](#).

Two new extension controls provide a mechanism to selectively enable markup and other XForms controls depending of the values stored in the XForms instance. The syntax is modeled after the XSLT language: `if` and `choose/when/otherwise`. See the [XForms Conditionals](#) documentation for more information.

The XForms element `<itemset>` is now supported. You can use it in conjunction with XForms selection controls (`<xforms:select` and `<xforms:select1>`) when you want the list of items the end-user has to pick from to come from the instance, instead of being listed in view. For more details, see the [XForms controls](#) documentation and the [Selection Controls](#) example.

You can now use the `ref` attribute on `<xforms:label>`, `<xforms:alert>`, `<xforms:help>`, and `<xforms:hint>` to have the displayed message come from the instance instead of being specified in the view. For more details, see the [Label, Alert, Help, and Hint](#) documentation.

You can now bind a submit control (`<xforms:submit>`) to a node in the instance with the `ref` attribute. If you do so, the submit control will only be displayed if the node it is bound to is relevant. This way you can define in the XForms model rules that specify when a submit control is to be displayed.

1.5.2. Standard Representation of Binary and Text Documents

In OPS [XPL](#) and [pipelines](#) only deal with XML documents. In order to handle non-XML data such as binary and text documents in pipelines, OPS defines [two standard XML document formats](#) to embed binary and text documents within an XML infoset. This solution has the benefit of keeping XPL simple by limiting it to pure XML infosets, while allowing XPL to conveniently manipulate any binary and text document.

These document formats are widely used by processors such as the [URL generator](#), the [HTTP serializer](#), the [Email processor](#), and [converters](#).

1.5.3. Improved Request Body and Uploaded Files Support

It was possible in the past with the [Request generator](#) to obtain the request body, which was automatically parsed as XML. This behavior has now been changed. It is instead possible to obtain the request body as `xs:anyURI` or `xs:base64Binary` ([standard binary document format](#)). Parsing may then be done using the [URL generator](#). In addition, the body can now be read several times. This change brings consistency with the handling of uploaded files, and the support for binary files throughout OPS.

In a related change, the [Request generator](#) now supports specifying whether uploaded files or the request body must be generated as `xs:anyURI` or `xs:base64Binary`.

1.5.4. URL Generator

The [URL generator](#) now supports streaming binary files encoded with Base64. The [XForms Upload example](#), among others, illustrates this capability.

The [URL generator](#) now supports reading text files. The [Employees example](#) illustrates this capability.

The [URL generator](#) now correctly detects as XML the following content types:

- `application/xml`, the recommended content type for XML
- `text/xml`, the now deprecated content type for XML
- content types ending with `+xml` such as `image/svg+xml`

1.5.5. Serializers

The pre-2.6 serializer architecture did not clearly separate the following concerns:

- Conversion of XML documents to a character or byte stream.
- Sending a character or text stream to a particular destination, such as a web browser through HTTP, or a file on disk.

The new architecture introduces [converters](#) in charge of the first task, and plain [serializers](#) in charge of the second task. It relies on the [standard representation](#) for binary and texts documents.

Backward compatibility for existing serializers is kept, but it is recommended to use the new converters and serializers whenever possible.

A new serializer, the [HTTP serializer](#), supports decoding of binary or text data, and sending the resulting stream to an HTTP client. The [XForms Upload example](#) illustrates this capability. The HTTP serializer can be used for example in conjunction with [converters](#) and the URL generator.

The [HTTP serializer](#) uses the recommended `application/xml` content type by default rather than `text/xml`.

The `omit-xml-declaration` and `standalone` configuration parameters of the [XML converter](#) are now documented. The `cache-control/use-local-cache` configuration parameter of the [HTTP serializer](#) is now documented.

1.5.6. Examples

The data access layer of the BizDoc application has been refactored to support the following backends:

- **XML databases:** XML documents are accessed through the XML:DB processors. Currently, the backend assumes the built-in open source eXist database.
- **SQL databases:** XML documents are accessed as a text or native XML format.

It is now possible to visualize each document as XML from the user interface.

The new [Employees example application](#) illustrates the following aspects of OPS:

- CRUD operations, paging and sorting with a SQL backend
- XForms controls and validation with XML Schema
- XForms upload control
- Reading plain text files
- Using the Java processor
- Exporting Comma-Separated Values (CSV) files
- Exporting and importing Excel files
- Calling and implementing Web Service
- Authentication (login / logout) and roles
- LDAP access

1.5.7. Page Flow

XSLT views in [Page Flow](#) can now access the XForms instance using `document('input:instance')` or `doc('input:instance')`. In the past, it was often necessary for the model to aggregate its output with the XForms instance to make both documents accessible from the view.

1.5.8. Email Processor

The [Email processor](#) features the following improvements:

- **Multipart hierarchy:** it is possible to have multiple levels of multipart messages.
- **Binary attachments:** binary files such as images and PDF files can be attached to an email.
- **Dynamic attachments:** attachments can be generated dynamically. It is for example possible to attach a dynamically generated chart or PDF file.

- **Character encoding:** character encoding can be specified for text content.

In addition, property naming has been revisited. The `test-smtp-host` and `test-to` replace the deprecated `host` and `forceto` properties.

1.5.9. Custom Processors

There is a new facility to declare custom processors that does not involve modifying the original `processors.xml` file. To do this, create a file called `oxf:/config/custom-processors.xml`. Its format is the same as the original `processors.xml`.

The `Processor` API has been enhanced to provide access to the inputs and outputs actually connected (as opposed to just those that have been declared). This provides access to optional inputs and outputs. Please refer to the Javadoc for the `Processor` class for more information, as well as the `ListInputs` unit test.

1.5.10. WebDAV Resource Manager

A [WebDAV resource manager](#) is now included. It supports basic authentication.

1.5.11. J2SE 5.0 Support

The source code of OPS now compiles with Java 5.

1.6. Installing Orbeon PresentationServer

1.6.1. Downloading

Orbeon PresentationServer (OPS) can be downloaded from <http://www.orbeon.com/community/downloads>.

1.6.2. System Requirements

To install OPS you need an application server that runs on Java version 1.4.2 (or later) and implements the Servlet API 2.3 (or later). OPS has been tested on the following application servers:

- Apache Tomcat 4.1.31 (JDK 1.4.2)
- Apache Tomcat 5.5.4 (JDK 1.5.0)
- BEA WebLogic Server 9.1 (JRockit)
- IBM WebSphere 6
- JOnAS 4.6.6 (Tomcat 5.5.12, JDK 1.5.0)
- JBoss 3.2.7 and 4.0

Please contact us if you have questions about support for other application servers or versions.

1.6.3. Installing OPS on Apache Tomcat

1. Assuming that `TOMCAT_HOME` represents the location of your Tomcat installation: create a new `TOMCAT_HOME/webapps/ops` directory.
2. Unzip `ops.war` in the `ops` directory you just created.
3. With Tomcat 5, move `xercesImpl.jar` and `xmlParserAPIs.jar` from `common/endorsed` to `server/lib`. This way Xerces will be available to Tomcat, but it won't override the version of Xerces and standard XML APIs that comes with OPS.
4. You can now start Tomcat, and access `http://localhost:8080/ops/` to test your installation (replacing `localhost` and `8080` with the host name and port number of your Tomcat installation if different from the default), or perform one of the optional installation steps below.
5. Optionally, to run the authentication example:
 1. Open `TOMCAT_HOME/webapps/ops/WEB-INF/web.xml` and uncomment the `security-constraint`, `login-config` and `security-role` declarations at the end of the file.
 2. Open `TOMCAT_HOME/conf/server.xml` and uncomment the following declaration: `<Realm className="org.apache.catalina.realm.MemoryRealm" />`
 3. Edit `TOMCAT_HOME/conf/tomcat-users.xml` and replace the content of this by with:

```
<tomcat-users>
  <role rolename="adminstrator"/>
  <user username="admin" password="password" roles="adminstrator"/>
</tomcat-users>
```

1.6.4. Installing OPS on BEA WebLogic 9.1

1. Select a directory where you want to store your web application. Let's assume the path you chose is `C:/WebApps/ops`.
2. Unzip `ops.war` into `C:/WebApps/ops`. There should now be a directory called `WEB-INF` under `C:/WebApps/ops`.
3. Start WebLogic's administration console.
4. Use the console to install a new Web application. When prompted to select a WAR file, point to the directory `C:/WebApps/ops`. When prompted for a context path, choose a value such as `ops`. Complete the installation and start the web application.

5. You should now be able to access the OPS examples by pointing your browser to the address of your WebLogic server followed by the context path you chose, for example: `http://localhost:7001/ops/`.

1.6.5. Installing OPS on BEA WebLogic 7.0 and 8.1

Warning

OPS 3.0 hasn't been tested with these versions of WebLogic, but you may want to try the following instructions for OPS 2.8.

1. Assume that `DOMAIN` represents your WebLogic domain directory (typically `c:\bea\user_projects`). Create a new directory: `DOMAIN\applications\orbeon`.
2. Unzip `ops.war` in the `orbeon` directory you just created.
3. Edit the `startWeblogic.cmd` (in `DOMAIN`) and change `set STARTMODE=true` to `set STARTMODE=false`. This starts WebLogic in development mode. In development mode, WebLogic automatically loads and deploys the content of the application directory. If you don't want to start the server in development mode, you have to explicitly declare a Web application in the `config.xml`.
4. To improve performance on WebLogic (**highly recommended!**):
 1. Start WebLogic (e.g. with `startWebLogic.cmd`)
 2. Make sure you can access the OPS examples with your browser (by going to `http://localhost:7001/orbeon/`)
 3. Stop WebLogic
 4. Open the `config.xml` file in an editor. Look for the `<WebAppComponent Name="orbeon">` element and add the attribute: `ServletReloadCheckSecs="-1"`. This will prevent WebLogic from checking if a servlet has changed in the application and will make OPS *much* faster.
5. Optionally, to run the authentication example:
 1. Open `DOMAIN/applications/orbeon/WEB-INF/web.xml` and uncomment the `security-constraint`, `login-config` and `security-role` declarations at the end of the file.
 2. Go to the WebLogic Console with a browser.
 3. Create a new user named `admin` with a password of your choice.
6. Once OPS is properly installed, you can start WebLogic as usual with the `startWeblogic.cmd` script (in `DOMAIN`).

1.6.6. Installing OPS on IBM WebSphere 6

1. Launch WebSphere server.
 - On Windows go to Control Panel, Administrative Tools, Services. Look for IBM WebSphere Application Server and make sure it is started.
 - On Linux/UNIX, assuming that `WSAS_HOME` represents the location of your WebSphere installation, run `WSAS_HOME/profiles/default/bin/startServer.sh server1`.
2. Log in to the administrative console.
 - Got to `http://localhost:9060/ibm/console/`.
 - The default administrator login is `admin`.
3. Install and deploy OPS (`ops.war`).
 - Click on Applications / Install New Application.
 - Select the `ops.war` to upload, choose a context path like `/ops` (from now on we will assume this was your choice).
 - Hit "next" until you get to the end of the wizard, then hit "finish". You can leave the defaults everywhere while going through the wizard.
 - Save the configuration.
 - Click on Applications / Enterprise Applications.
 - Select `ops_war` and click on the "start" button.
4. Run and modify the examples.
 - Go to `http://localhost:9080/ops/`.
 - You can view the log from OPS in `WSAS_HOME/profiles/default/logs/server1/SystemOut.log`.
 - You can modify the examples resources as the application sever is running and see the results of your modifications on the fly. The resources are stored under `WSAS_HOME/profiles/default/installedApps/yourmachineNode01Cell/orbeon_war.ear/ops.war/WEB-INF/resources`. For instance, try to modify `examples/tutorial/hello1/view.xhtml`: replace "Hello World!" by your own message, and reload the page in the browser to see the result.

1.6.7. Installing OPS on JBoss 3.2.7 and 4.0

1. Assuming that `JBOSS_HOME` represents the location of your JBoss installation: create a new `JBOSS_HOME/server/default/deploy/ops.war` directory.
2. Unzip `ops.war` in the `ops.war` directory you just created.
3. If you are using JBoss 4.0, edit `JBOSS_HOME/server/default/deploy/jbossweb-tomcat55.sar/META-INF/jboss-service.xml` and change the value in `<attribute name="UseJBossWebLoader">` from `false` to `true`.
4. Start JBoss by running `JBOSS_HOME/bin/run.bat` (or `run.sh` on UNIX).

5. Run an modify the examples.

- Go to `http://localhost:8080/ops/`
- You can modify the examples resources as the application sever is running and see the results of your modifications on the fly. The resources are stored under `JBOSS_HOME/server/default/deploy/ops.war/WEB-INF/resources`.

6. Optionally, to run the authentication sample:

- Open `JBOSS_HOME/server/default/deploy/ops.war/WEB-INF/web.xml` and uncomment the `security-constraint`, `login-config` and `security-role` declarations at the end of the file.
- Open `JBOSS_HOME/server/default/deploy/ops.war/WEB-INF/jboss-web.xml` and uncomment the `security-domain` element near the end of bottom of the file.
- Open `JBOSS_HOME/server/default/conf/login-config.xml` and add the following application policy to the list of policies :

```
<appl i cati on- pol i cy name="orbeon-demo">
  <authenti cati on>
    <l ogi n- modul e code="org.j boss. securi ty. auth. spi . UsersRol esLogi nModul e" fl ag="requi red">
      <modul e- opti on name="usersProperti es">orbeon-demo-users. properti es</modul e- opti on>
      <modul e- opti on name="rol esProperti es">orbeon-demo-rol es. properti es</modul e- opti on>
    </l ogi n- modul e>
  </authenti cati on>
</appl i cati on- pol i cy>
```

1.6.8. Security

For security reasons, you might want to run OPS under a Security Manager. Java's Security Manager allows you to control the Java sandbox and which resources the application can access. When installed correctly, the Security Manager can prevent unauthorized code to execute malicious actions, such as deleting files on the server or initializing network connections. For more information, please read the [Security in Java 2 SDK 1.2](#) tutorial and the [Security Manager API](#).

Follow the steps below to install the Security Manager:

1. Download the `policy` file.
2. Append the permissions to the application server policy file. The table lists the policy file for the supported servers.

Apache Tomcat	catalina.policy
BEA Weblogic	weblogic.policy
IBM WebSphere	was.policy
Sun ONE	server.policy

3. Add the following system properties to the server startup script.
 - `oxf.home`: Location of the OPS exploded WAR file
 - `oxf.resources`: Location of OPS resources directory
4. Modify the startup script to enable the security manager. Add the following system properties:
 - `-Djava.security.manager`
 - `-Djava.security.policy=="path to the policy file"`

2. Core Technologies Reference

2.1. XForms Reference

2.1.1. Scope

Web applications use forms to collect data from users. Orbeon PresentationServer (OPS)'s form handling capabilities are based on [XForms](#), namely the [XForms 1.0 W3C Recommendation](#). This section provides an introduction to XForms concepts and explains how to use XForms in your OPS application.

Note

This document is considered a work in progress. While it does cover some generic features of XForms, it focuses before all on features specific to the OPS XForms engine. For more information about XForms, please refer to the following resources:

- [XForms 1.0 \(second edition\) W3C Proposed Edited Recommendation](#)
- [XForms Essentials](#), by Micah Dubinko (also at [Amazon](#)).
- [XForms: XML Powered Web Forms](#), by T. V. Raman (also at [Amazon](#)).

2.1.2. Introduction to XForms

XForms 1.0 has been designed by the W3C based on experience with HTML forms. It was promoted to the rank of W3C Recommendation in October 2003, and a [second edition](#) of the specification has been released in October 2005. As of December 2005, mainstream browsers (Internet Explorer, Mozilla / Firefox, Opera, Safari) do not support XForms natively, although XForms support in Mozilla is under way and plugins are available for Internet Explorer. However you can already leverage the benefits of XForms today by using a hybrid client-side / server-side XForms engine like the one provided in OPS. The OPS XForms engine transparently generates HTML forms and performs the work that would be done by an XForms-compliant browser. This way you can start leveraging XForms today, be ready for upcoming XForms-compliant browsers, and work smoothly with the mainstream browsers that are deployed in the marketplace.

For more information about the whys and therefores of server-side XForms, please read our article, [Are Server-Side XForms Engines the Future of XForms?](#) ([pre-conference version](#) and [updated version](#)).

Compared to HTML forms, XForms offers a higher level approach to forms. The benefits are that less programming is needed (less JavaScript, and less server-side programming), so forms are easier to create and modify. As an illustration, let's consider some facets of XForms:

1. **XML Representation of Forms.** XForms clearly defines how data entered by the end-user is collected: it is stored in an XML document called an XForms *instance*, an initially empty, "skeletal" XML instance document that defines the structure of the data you wish to collect from the user, which is afterwards filled out with information collected from the user. For example, credit card information collected on a web site can be structured as follows:

```
<credi t-card>
  <type/>
  <number/>
  <expi rati on-month/>
  <expi rati on-year/>
</credi t-card>
```

The outcome of the user filling out a form collecting this information could be this complete XML document:

```
<credi t-card>
  <type>vi sa</type>
  <number>1234567812345678</number>
  <expi rati on-month>8</expi rati on-month>
  <expi rati on-year>2008</expi rati on-year>
</credi t-card>
```

An application using this data to do some processing (e.g. checking the validity of the credit card) receives the above XML document. There is no need to write code to go read HTTP request parameters, or to use a framework performing this task: XForms does it all.

2. **Declarative Constraints and Validation.** More often than not, there are constraints on the data that can be entered by the end-user. For instance, in the example we just considered, the card number must have 16 digits and the expiration month must be a number between 1 and 12. Traditionally code must be written to check for those constraints. And more code must be written to handle error conditions (getting back to the page displaying the form and showing the appropriate error messages). All this is done in a very simple and declarative way with XForms. For instance, checking that the expiration month is valid number between 1 and 12 can be done with:

```
<xforms:bi nd nodeset="/credi t-card/expi rati on-month" type="xs:i nteger" constrai nt=". >= 1 and 12 >= ." />
```

An error message can be attached to the "month" text field and if the end-user enters an invalid month the XForms engine will notice that the above constraint is not met and will display the error message. You do not have to write any code for this to happen. We will see later how you go about doing this with XForms in more details.

3. **Declarative Event Handling.** User interfaces need to react to user event such as mouse clicks and character entry. With most UI frameworks, developers must register event handlers and implement them in JavaScript, Java, or other traditional imperative languages. With XForms, a set of predefined event handlers and actions are available, which cover a set of useful cases without requiring understanding the complex syntax and semantic of JavaScript or Java. For example, to set a value into an XForms instance, you write:

```
<xforms: setvalue ref="/credit-card/expiration-month">11</xforms: setvalue>
```

Once you have learned the simple built-in XForms actions, you can combine them in sequences to obtain more complex behavior.

2.1.3. Getting Started With the OPS XForms Engine

The easiest way to get started with simple examples is to use the OPS XForms Sandbox. This tool allows you to upload example XForms files from your web browser and to see the results directly. You can access the XForms sandbox:

- **Online:** visit [this link](#) to access the online public XForms Sandbox.
- **Locally:** if this documentation is produced by your local installation of OPS, visit [this link](#).

After submitting an XHTML + XForms file, the result, or errors, should display. If you have changed your local XForms file, reloads that page in your browser and this will upload again your local XForms file and the XForms Sandbox will run the new version. To select another file to upload use your browser quotes "back" button to return to the main XForms sandbox page.

2.1.4. Programming With XForms 1.0

To help in our exploration of XForms we consider a specific example: an XForms Credit Card Verifier. This example displays a simple form asking for a credit card number and related information to be entered, as shown on the screenshot to the right. The information entered by the end-user is validated by a set of rules and errors are flagged in red.

First, the information contained in the form is stored in an XML document called an *XForms instance*, which is the skeleton or shell that will contain the data captured by the form. You define an XForms instance within an `xforms:instance`. In the Credit Card Verifier the unique XForms instance is declared with:

The screenshot shows a web form titled "Please correct the errors on this page." with a red minus icon. Below the title is a list of errors: "No alphabetical characters or signs are allowed in credit card number". The form fields are: "Card type:" with a dropdown menu showing "Visa"; "Number:" with a text input containing "abc" and a red error icon; "Expiration month:" with a text input containing "11"; "Expiration year:" with a text input containing "2005"; "Verification code:" with an empty text input; "Card valid:" with a label "false"; and a "Verify" button.

```
<xforms: instance id="credit-card-instance">
  <credit-card>
    <type/>
    <number/>
    <expiration-month/>
    <expiration-year/>
    <verification-code/>
    <valid/>
  </credit-card>
</xforms: instance>
```

The XForms instance does not have to be empty of data: it can contain initial values for the form. Here we set the `valid` element to the value "false" by default:

```
<xforms: instance id="credit-card-instance">
  <credit-card>
    <type/>
    <number/>
    <expiration-month/>
    <expiration-year/>
    <verification-code/>
    <valid>false</valid>
  </credit-card>
</xforms: instance>
```

XForms instances are always contained in an *XForms model*, which:

1. Declares one or more XForms instance.
2. Optionally, declares a set of rules attached to the XForms instances.
3. Optionally, declares submissions.

At a minimum, the XForms instance above must be encapsulated as follows:

```
<xforms:model id="main-model">
  <xforms:instance id="credit-card-instance">
    <credit-card>
      <type/>
      <number/>
      <expiration-month/>
      <expiration-year/>
      <verification-code/>
      <valid>false</valid>
    </credit-card>
  </xforms:instance>
</xforms:model>
```

Note that instances and models can have an optional `id` attribute. If you have only one model and one instance, the `id` is optional, but it becomes very convenient when more than one model or instance are used.

In addition to one or more XForms instances, an XForms model can declare a set of "rules", called "model item properties". Let's write a set of rules for the above Credit Card Validation form. Specifically we want to:

1. Check that the credit card number is a number and valid according to particular credit card rules
2. Check that the expiration month is valid (integer between 1 and 12)
3. Check that the expiration year is valid (4 digit number)
4. Display the "verification code" line only if the card type is Visa or MasterCard
5. Check that the verification code is valid only for Visa or MasterCard

You describe each one of those rules with an `<xforms:bind>` element in the XForms model. Rules apply to elements and attributes in the XForms instance. You specify the elements and attributes each rule applies to with an XPath expression in the mandatory `nodeset` attribute. In addition to the `nodeset` attribute you want to have at least one attribute specifying the essence of the rule. We go over all the possible attributes later in this section, but first let's see how we can express the above rules for the Credit Card Verifier form:

1. You specify that the credit card number must be a number with:

```
<xforms:bind nodeset="number" type="xs:integer"/>
```

The value of the `type` attribute is a W3C XML Schema simple type. You can see the list of simple types in the [XML Schema primer](#). If the end-user enters an invalid credit card number (i.e. not a number), an error will be displayed as shown in the screenshot on the right.

2. You can also constrain the value of an element or attribute with an XPath expression in the `constraint` attribute. For instance you specify that the expiration month must be an integer between 1 and 12 with:

```
<xforms:bind nodeset="expiration-month" constraint=". castable as xs:integer and . >= 1 and 12 >= ." />
```

Note that we have decided here not to bother checking the expiration month if no credit card number was entered.

3. Similarly, you check that the expiration year is a 4 digit number with:

```
<xforms:bind nodeset="expiration-year" constraint=". castable as xs:integer and string-length(.) = 4" />
```

4. You hide the "verification code" text field for American Express cards with:

```
<xforms:bind nodeset="verification-code" relevant=".. /type = 'visa' or .. /type = 'mastercard' " />
```

The attribute we use here is `relevant`. By default, everything is relevant in the XForms instance. If a "relevant" rule is specified, the XPath expression is evaluated for each node in the `nodeset`, and if the expression returns false, then the node is not considered relevant. When a node is not relevant, the corresponding widget is not displayed (more on this later).

5. Finally, you check that the verification code is entered for Visa and Mastercard:

```
<xforms:bind nodeset="verification-code" constraint="/credit-card/type = ('visa', 'mastercard') and . castable as xs:positiveInteger" />
```

Because the `verification-code` element has both a `relevant` and a `constraint` attribute, we combine them on the same `xforms:bind`:

```
<xforms:bind nodeset="verification-code" relevant=".. /type = 'visa' or .. /type = 'mastercard' " constraint="/credit-card/type = ('visa', 'mastercard') and . castable as xs:positiveInteger" />
```


XPath expressions in `xforms:bind` are by default relative to the root element of the first XForms instance. This allows you to write the first constraint above:

- Relatively to the root element of the first XForms instance:

```
<xforms:bind nodeset="number" type="xs:integer"/>
```

- With an absolute path in the first XForms instance:

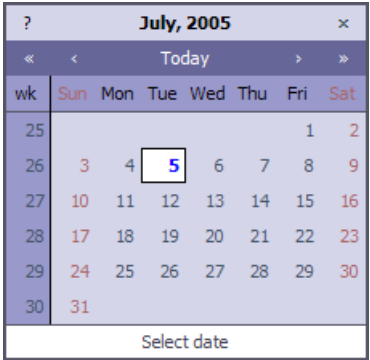
```
<xforms:bind nodeset="/credit-card/number" type="xs:integer"/>
```

- Referring explicitly to the "credit-card-instance" using the `instance()` function:

```
<xforms:bind nodeset="instance('credit-card-instance')/number" type="xs:integer"/>
```

Now that we have seen a few examples of model item properties, let's go over all the XForms model item properties. Model item properties can essentially be used for 3 purposes:

Validation	<p>The purpose of validation is to determine if the content of an element or attribute in the XForms instance is valid. Invalid values can have an impact on how a form is displayed (you might want to highlight errors and show some information to help the end-user to correct the issue). Also, the XForms engine makes sure that invalid data cannot be submitted. There are 3 ways to validate the content of an element or attribute:</p> <ul style="list-style-type: none">■ required — You can specify in the <code>required</code> attribute an XPath expression that determines if a value is required. The XPath can be as simple as <code>true()</code>, or more complex and depend on other values entered by the end-user. By default values are not required.■ type — In the <code>type</code> attribute you can specify a W3C XML Schema simple type. The <code>type</code> attribute complements the <code>required</code> attribute, but applies separately. <p>In Addition, some XML schema types have special behavior:</p> <ul style="list-style-type: none">■ xs:date — The input field is complemented by a pop-up calendar. The user can enter a date manually, or use the calendar to select a date in the past or in the future. The calendar is customizable by the application developer under: <code>oxf:/config/theme/jscalendar</code>■ constraint — The <code>constraint</code> attribute supports any XPath expression that returns a boolean value. If <code>false()</code> is returned, then the value is considered invalid, otherwise it is considered valid.
Calculation	<p>The purpose of calculations is to dynamically compute values. You do this with the <code>calculate</code> attribute:</p> <ul style="list-style-type: none">■ calculate — The content of the element or attribute will be set to the result of the evaluation of the XPath expression in the <code>calculate</code> attribute. This way you can automatically compute some values in the XForms instance based on other values, typically entered by the end-user. By default, nodes that contain calculated values are read-only.
Visibility	<p>In general XForms instance nodes are not read-only and are relevant, which means that if an XForms control is bound to that node (e.g. a text field), the control is displayed and is editable by the end-user. You can change this by providing XPath expressions in the <code>readonly</code> and <code>relevant</code> attributes:</p> <ul style="list-style-type: none">■ readonly — If the XPath expression in <code>readonly</code> evaluates to true, the control will be displayed in non-editable mode. Typically, in an XHTML user interface only the current value is displayed, instead of displaying a form element, like a text field.■ relevant — If the XPath expression in <code>relevant</code> evaluates to false, the control will not be displayed at all.



XForms controls are similar to HTML form elements: they include text fields, drop down lists, checkboxes, etc. These are some differences between HTML forms elements and XForms controls:

- The value displayed by an XForms control comes from a node of the XForms instance. When you declare a control, you bind it to a node of your XForms instance with an XPath expression in the `ref` attribute. For instance this text field a text field is bound to the `<number>` element, which a child of `<credit-card>`:

```
<xforms:input ref="/credit-card/number"/>
```

- The way a control is rendered depends on model item properties that apply to the node the control is bound to: if it is bound to an invalid node then an error can be displayed; if the control is bound to a read-only node the value is displayed in read-only mode; if the node is not relevant the control isn't be displayed at all; if the control is bound to a non-existing node, the control is considered non-relevant and is not displayed;

The table below lists all the available XForms controls and shows for each one the XML you need to use in your view, as well as an example showing that control in action.

Control	XForms in the view	Example
Text field 	<pre><xforms: input ref="text" /></pre>	XForms Controls
Password field 	<pre><xforms: secret ref="secret" /></pre>	XForms Controls
Text area 	<pre><xforms: textarea ref="textarea" /></pre>	XForms Controls
Radio buttons 	<pre><xforms: select1 ref="carrier" appearance="full" > <xforms: i tem> <xforms: l abel >Fedex</xforms: l abel > <xforms: val ue>fedex</xforms: val ue> </xforms: i tem> <xforms: i tem> <xforms: l abel >UPS</xforms: l abel > <xforms: val ue>ups</xforms: val ue> </xforms: i tem> </xforms: select1></pre>	XForms Controls
Single-selection lists 	<pre><xforms: select1 ref="carrier" appearance="compact"> <xforms: i tem> <xforms: l abel >Fedex</xforms: l abel > <xforms: val ue>fedex</xforms: val ue> </xforms: i tem> <xforms: i tem> <xforms: l abel >UPS</xforms: l abel > <xforms: val ue>ups</xforms: val ue> </xforms: i tem> </xforms: select1></pre>	XForms Controls
Combo box 	<pre><xforms: select1 ref="payment" appearance="minimal" > <xforms: i tem> <xforms: l abel >Cash</xforms: l abel > <xforms: val ue>cash</xforms: val ue> </xforms: i tem> <xforms: i tem> <xforms: l abel >Credi t</xforms: l abel > <xforms: val ue>credi t</xforms: val ue> </xforms: i tem> </xforms: select1></pre>	XForms Controls
Checkboxes	<pre><xforms: select ref="wrappi ng" appearance="full" > <xforms: choi ces> <xforms: i tem> <xforms: l abel >Hard-box</xforms: l abel > <xforms: val ue>box</xforms: val ue> </xforms: i tem> <xforms: i tem></pre>	

	<pre> <xforms:label>Gift</xforms:label> <xforms:value>gift</xforms:value> </xforms:item> </xforms:choices> </xforms:select> </pre>	
List 	<pre> <xforms:select ref="taste" appearance="compact"> <xforms:item> <xforms:label>Vanilla</xforms:label> <xforms:value>vanilla</xforms:value> </xforms:item> <xforms:item> <xforms:label>Strawberry</xforms:label> <xforms:value>strawberry</xforms:value> </xforms:item> </xforms:select> </pre>	XForms Controls
Trigger button 	<pre> <xforms:trigger> <xforms:label>Add carrier</xforms:label> </xforms:trigger> </pre>	XForms Controls
Submit button 	<pre> <xforms:submit submission="main-submission"> <xforms:label>Submit</xforms:label> </xforms:submit> </pre>	-
Submit link 	<pre> <xforms:submit submission="main-submission" appearance="xforms:link"> <xforms:label>Submit</xforms:label> </xforms:submit> </pre>	-
Submit image 	<pre> <xforms:submit submission="main-submission" appearance="xforms:image"> <xxforms:img src="images/submit.gif"/> <xforms:label>Submit</xforms:label> </xforms:submit> </pre>	-
Upload 	<pre> <xforms:upload ref="files/file[1]"> <xforms:filename ref="@filename"/> <xforms:mediatype ref="@mediatype"/> <xxforms:size ref="@size"/> </xforms:upload> </pre>	Upload Control
Range 	<pre> <xforms:range ref="range/value"> <xforms:send submission="countries-submission" ev:event="xforms- value-changed"/> </xforms:range> </pre>	XForms Controls


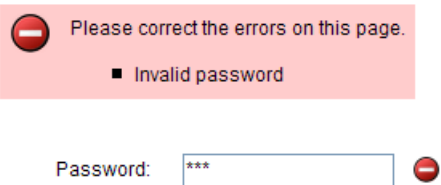

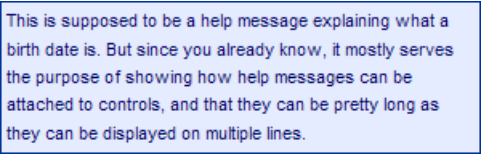
In the examples above, the labels and values for the select and select1 controls are declared in the control element with multiple `<xforms:item>` elements. Alternatively the label/value pairs can be pulled out from the instance. You do this with an `<xforms:itemset>` element (instead of `<xforms:item>` elements):

```

<xforms:select1 ref="country" appearance="compact">
  <xforms:itemset nodeset="instance('countries')/country">
    <xforms:label ref="name"/>
    <xforms:value ref="us-code"/>
  </xforms:itemset>
</xforms:select1>

```

Nested inside each XForms control element, you can specify additional elements that can alter the way the control is displayed. The table below lists those elements:

Label		By default a label is used in submit controls, as well as the single and multiple selection controls, as shown in the table above. The <code>label</code> element is mandatory for all controls.
Alert		In each control you can specify an error message that can be displayed if the value entered by the user triggers a validation error. <pre><xforms: secret ref="secret"> <xforms: alert>Invalid password</xforms: alert> </xforms: secret></pre>
Hint		You can specify a hint on each control, which is displayed next to the control and becomes highlighted when the control is selected. <pre><xforms: textarea ref="textarea"> <xforms: hint>Enter at least 11 characters</ xforms: hint> </xforms: textarea></pre>
Help		If you specify a help message for a control, an icon with a question mark is displayed next to the control. A pop-up shows the help message when you position the mouse cursor over the icon. <pre><xforms: input ref="date" class="xforms-date"> <xforms: label class="fixed-width">Birth date: </ xforms: label> <xforms: help> This is supposed to be a help message explaining what a birth date is. But since you already know, it mostly serves the purpose of showing how help messages can be attached to controls, and that they can be pretty long as they can be displayed on multiple lines. </xforms: help> </xforms: input></pre>

In the examples above, the text displayed is directly in the `<xforms:label>`, `<xforms:alert>`, `<xforms:help>`, or `<xforms:hint>` element. Alternatively that text can come from an XForms instance with a `ref` attribute on any one of those elements. The `ref` references a node in the instant that contains the text to use. This is illustrated in the code below from the [XForms Text Controls](#) example:

```
<xforms: secret ref="secret">
  <xforms: alert ref="@alert"/>
</xforms: secret>
```

XForms allows you to upload files with the XForms Upload control:

```
<xforms: upload ref="files/file[1]">
  <xforms: filename ref="@filename"/>
  <xforms: mediatype ref="@mediatype"/>
  <xxforms: size ref="@size"/>
</xforms: upload>
```

The related section of the XForms instance can look like this:

```
<files>
  <file filename="" mediatype="" size="" xsi:type="xs:anyURI"/>
</files>
```

The `file` element is the element storing the result of the file upload. The result can be stored in two ways:

- As a URL, by specifying the type `xsi:anyURI`.
- As Base64-encoded text, by specifying the type `xsi:base64Binary`. Base64 is a mechanism to encode any binary data using a 65-character subset of US-ASCII. Using this mechanism allows embedding binary data into XML documents, at the typical cost of taking 50% more space than the original binary data. For more information, please refer to the [RFC](#).

Note

It is mandatory to specify either one of `xs:anyURI` or `xs:base64Binary`.

The optional `xforms:filename`, `xforms:mediatype`, and `xxforms:size` (the latter being an extension) allow storing metadata about an uploaded file:

- `xforms:filename`: stores the file name sent by the user agent
- `xforms:mediatype`: store the media type sent by the user agent
- `xxforms:size`: stores the actual size in bytes of the uploaded data

Note that the file name and the media type are provided by the user agent (typically a web browser) and are not guaranteed to be correct.

The result of a file upload can look as follows when using `xs:anyURI`:

```
<file filename="photo.jpg" mediatype="image/jpeg" size="2345" xsi:type="xs:anyURI">file:/C:/Tomcat/temp/
upload_00000005.tmp</file>
```

Warning

The URL stored as the value of the upload is temporary and only valid for the duration of the current request. It is only accessible from the server side, and will not be accessible from a client such as a web browser. It is not guaranteed to be a `file:` URL, only that it can be read with Presentation Server's URL generator.

The contents of the file can be retrieved using the URL Generator. The result will be an XML document containing a single root element containing the uploaded file in Base64-encoded text.

Note

Using the `xs:anyURI` type allows OPS to make sure the uploaded file does not have to reside entirely in memory. This is the preferred method for uploading large files.

The result of a file upload can look as follows when using `xs:base64Binary`:

```
<file filename="photo.jpg" mediatype="image/jpeg" size="2345" xsi:type="xs:base64Binary">
/9j/4AAQSkZJRgABAQEASABIAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwshBwYGBw4KCggLEA4RERA0
EA8SFBoWEhMYEw8QFh8XGBsbHR0dERYgIh8cIhocHRz/2wBDAQUBQcGBwOHBwOcEhASHBwCHBwc...
</file>
```

In this case, the uploaded file is encoded and directly embedded into the XML instance. This is a good method to handle small files only, because the entire file is converted and stored in memory.

Make sure, in your XForms model, that you have the correct submission method and encoding:

```
<xforms:submission method="post" encoding="multipart/form-data" xmlns:xforms="http://www.w3.org/2002/xforms"/>
```

A very common requirement of user interfaces consists in repeating visual elements, such as rows in a table or entries in a list. Those repeated sections usually have an homogeneous aspect: they all have the same or a very similar structure. For example, multiple table rows will differ only in the particular content they display in their cells. An example of this is an invoice made of lines with each a description, unit price, and quantity.

XForms provides a very powerful mechanism to implement such repeated structures: the `xforms:repeat` element. You use `xforms:repeat` around XHTML elements or XForms controls. For example, to repeat a table row, you write:

```
<xforms:repeat>
  <xhtml:tr>
    ...
  </xhtml:tr>
</xforms:repeat>
```

This is not enough to be functional code: you need to indicate to the `xforms:repeat` element how many repetitions must be performed. This is done not by supplying a simple count value, but by binding the element to a node-set with the `nodeset` attribute. Consider the following XForms instance:

```
<xforms:instance id="employees-instance" xmlns:xforms="http://www.w3.org/2002/xforms">
  <employees>
    <employee>
      <first-name>Alice</first-name>
    </employee>
    <employee>
      <first-name>Bob</first-name>
    </employee>
    <employee>
      <first-name>Marie</first-name>
    </employee>
  </employees>
</xforms:instance>
```

```
</empl oyees>
</xforms: i nstance>
```

Assuming you want to produce one table row per employee, add the following `nodeset` attribute:

```
<xforms: repeat nodeset="i nstance(' empl oyees-i nstance')/empl oyee">
  <xhtml: tr>
    ...
  </xhtml: tr>
</xforms: repeat>
```

This produces automatically three `xhtml:tr` rows. Note that we explicitly use the XForms `instance()` function, but you may not have to do so if that instance is already in scope. Then you display in each row the content of the `first-name` element for each employee:

```
<xforms: repeat nodeset="i nstance(' empl oyees-i nstance')/empl oyee">
  <xhtml: tr>
    <xhtml: td>
      <xforms: output ref="fi rst-name"/>
    </xhtml: td>
  </xhtml: tr>
</xforms: repeat>
```

This works because for each iteration, the *context node* for the `ref` attribute changes: during the first iteration, the context node is the first `employee` element of the XForms instance; during the second iteration, the second `employee` element, and so on.

Note

The `nodeset` attribute of `xforms:repeat` must point to a so-called *homogeneous collection*. Such a collection must consist of contiguous XML elements with same name and same namespace. XForms does not predict what happens if the node-set is not homogenous.

`xforms:repeat` may be used purely for display purposes, but it can also be used for interactively editing repeated data. This includes allowing the user to delete and insert rows. Two XForms actions are used for this purpose: `xforms:delete` and `xforms:insert`.

`xforms:delete` is provided with a `nodeset` attribute pointing to the homogenous collection into which the insertion must take place. It also has an `at` attribute, which contains an XPath expression returning the index of the element to delete. See how `xforms:delete` is used in these 3 scenarios:

```
<!-- This deletes the last element of the collection -->
<xforms:del ete nodeset="empl oyees" at="last()"/>
<!-- This deletes the first element of the collection -->
<xforms:del ete nodeset="empl oyees" at="1"/>
<!-- This deletes the currently selected element of the collection (assuming the repeat id 'employee-repeat') -->
<xforms:del ete nodeset="empl oyees" at="i ndex(' empl oyee-repeat')"/>
```

`xforms:insert` has a `nodeset` attribute pointing to the homogenous collection into which the insertion must take place. `xforms:insert` then considers the *last* element of that collection (and all its content if any) as a *template* for the new element to insert: it duplicates it and inserts it into the homogenous collection at a position you specify. The last element of an homogeneous collection therefore always acts as a *template* for insertions:

```
<!-- This inserts a copy of the template before the last element of the collection -->
<xforms: i nsert nodeset="empl oyees" at="last()" posi ti on="before"/>
<!-- This inserts a copy of the template after the last element of the collection -->
<xforms: i nsert nodeset="empl oyees" at="last()" posi ti on="after"/>
<!-- This inserts a copy of the template before the first element of the collection -->
<xforms: i nsert nodeset="empl oyees" at="1" posi ti on="before"/>
<!-- This inserts a copy of the template after the first element of the collection -->
<xforms: i nsert nodeset="empl oyees" at="1" posi ti on="after"/>
<xforms: i nsert nodeset="empl oyees" at="last()" posi ti on="after"/>
<!-- This inserts a copy of the template before the currently selected element of the collection -->
<xforms: i nsert nodeset="empl oyees" at="i ndex(' empl oyee-repeat')" posi ti on="before"/>
<!-- This inserts a copy of the template after the currently selected element of the collection -->
<xforms: i nsert nodeset="empl oyees" at="i ndex(' empl oyee-repeat')" posi ti on="after"/>
```

The `at` attribute contains an XPath expression returning the index of the element before or after which the insertion must be performed. The `position` element contains either `after` or `before`, and specifies whether the insertion is performed before or after the element specified by the `at` attribute.

It is important to note that while it is possible to delete the last element of an homogeneous collection, it becomes then impossible to insert a new element into that collection with XForms 1.0, since there is no longer a template element available in this case (save for using an XML submission with `replace="instance"`). This means that in general you will want to have at least one element in your collections.

In case you want the user interface to visually appear empty empty when there is "no more" elements in the collection, you can use the tip provided below, which can be used in most situations. The idea is to consider that the last element of the collection is never displayed, but always used as a template for `xforms:insert`:

```
<xforms: i nstance i d="empl oyees-i nstance" xml ns: xforms="http://www.w3.org/2002/xforms">
  <empl oyees>
    <empl oyee>
      <fi rst-name>Al i ce</fi rst-name>
```

```

    </empl oyee>
    <empl oyee>
      <fi rst-name>Bob</fi rst-name>
    </empl oyee>
    <empl oyee>
      <fi rst-name>Mari e</fi rst-name>
    </empl oyee>
    <!-- This is a template used by xforms:insert -->
    <empl oyee>
      <fi rst-name/>
    </empl oyee>
  </empl oyees>
</xforms: i nstance>

```

You do not want to display that template, however. Therefore you use an `xforms:repeat` element of the form:

```

<xforms: repeat nodeset="i nstance(' empl oyees-i nstance')/empl oyee[po si ti on() &lt; i; l ast()]">
  ...
</xforms: repeat>

```

The `position() < i; l ast()` condition tells `xforms:repeat` to consider all the elements of the collection except the last one. This causes the repetition to display zero iteration when there is one element in the collection, one iteration when there are two, etc. The `xforms:insert` action, on the other hand, operates on the entire collection including the last element, so that that element can be duplicated:

```

<xforms: i nsert nodeset="empl oyees" at="..." po si ti on="..." />

```

Another solution involves using an `xforms:bind` element which makes the last element of the collection non-relevant. This achieves the same result, but requires extra code, so the tip above is usually preferred.

Upon submission, some care must be taken with repeat template. For example, if the `first-name` element above is required, and the template contains an empty value as above, submission will fail. `xform:bind` statements must then also exclude the last element of the collection:

```

<xforms: bi nd nodeset="empl oyee[po si ti on() &lt; i; l ast()]/fi rst-name" requi red="true()" />

```

Note

If you are dealing with an XML document format which requires removing the last element of a collection, you have to post-process your XForms instance to remove such extra elements, and pre-process it to add such elements when initializing your XForms instance.

Insertions and deletions are typically performed when the user of the application presses a button, with the effect of adding a new repeated element before or after the currently selected element, or of deleting the currently selected element. You use an `xforms:trigger` control and the XPath `index()` function for that purpose:

```

<xforms: tri gger>
  <xforms: l abel >Add</xforms: l abel >
  <xforms: acti on ev: event="DOMActi vate">
    <xforms: i nsert nodeset="empl oyees" at="i ndex(' empl oyee-repeat' )" po si ti on="after" />
  </xforms: acti on>
</xforms: tri gger>

```

or:

```

<xforms: tri gger>
  <xforms: l abel >Del ete</xforms: l abel >
  <xforms: acti on ev: event="DOMActi vate">
    <xforms: del ete nodeset="empl oyees" at="i ndex(' empl oyee-repeat' )" />
  </xforms: acti on>
</xforms: tri gger>

```

Note that we use `xforms:action` as a container for `xforms:insert` and `xforms:delete`. Since there is only one action to execute, `xforms:action` is not necessary, but it may increase the legibility of the code. It is also possible to write:

```

<xforms: tri gger>
  <xforms: l abel >Add</xforms: l abel >
  <xforms: i nsert ev: event="DOMActi vate" nodeset="empl oyees" at="i ndex(' empl oyee-repeat' )" po si ti on="after" />
</xforms: tri gger>

```

or:

```

<xforms: tri gger>
  <xforms: l abel >Del ete</xforms: l abel >
  <xforms: del ete ev: event="DOMActi vate" nodeset="empl oyees" at="i ndex(' empl oyee-repeat' )" />
</xforms: tri gger>

```

Notice in that case how `ev:event="DOMActivate"` has been moved from the enclosing `xforms:action` to the `xforms:insert` and `xforms:delete` elements.

It is often desirable to nest repeat sections. Consider the following XForms instance representing a company containing departments, each containing a number of employees:

```
<xforms:instance id="departments">
  <departments>
    <department>
      <name>Research and Development</name>
      <employees>
        <employee>
          <first-name>John</first-name>
        </employee>
        <employee>
          <first-name>Mary</first-name>
        </employee>
      </employees>
    </department>
    <department>
      <name>Support</name>
      <employees>
        <employee>
          <first-name>Anne</first-name>
        </employee>
        <employee>
          <first-name>Mark</first-name>
        </employee>
        <employee>
          <first-name>Sophie</first-name>
        </employee>
      </employees>
    </department>
  </departments>
</xforms:instance>
```

This document clearly contains two nested sections subject to repetition:

- **Departments:** a node-set containing all the `department` elements can be referred to with the following XPath expression: `instance ('departments')/department`.
- **Employees:** a node-set containing all the `employee` elements can be referred to with the following XPath expression: `instance ('departments')/department/employees/employee`. However, if the *context node* of the XPath expression points to a particular department element, then the following *relative* XPath expression refers to all the `employee` elements under that department element: `employees/employee`.

Following the example above, here is how departments and employees can be represented in nested tables with XForms:

```
<xhtml:table>
  <xforms:repeat nodeset="instance('departments')/department">
    <xhtml:tr>
      <xhtml:td>
        <xforms:output ref="name"/>
      </xhtml:td>
      <xhtml:td>
        <xhtml:table>
          <xforms:repeat nodeset="employees/employee">
            <xhtml:tr>
              <xhtml:td>
                <xforms:output ref="first-name"/>
              </xhtml:td>
            </xhtml:tr>
          </xforms:repeat>
        </xhtml:table>
      </xhtml:td>
    </xhtml:tr>
  </xforms:repeat>
</xhtml:table>
```

In the code above, the second `xforms:repeat`'s `nodeset` expression is interpreted relatively to the `department` element of the parent `xforms:repeat` for each iteration of the parent's repetition. During the first iteration of the parent, the "Research and Development" department is in scope, and `employees/employee` refers to the two employees of that department, John and Mary. During the second iteration of the parent, the "Support" department is in scope, and `employees/employee` refers to the three employees of that department, Anne, Mark and Sophie.

There are two ways of providing the value to set with `<xforms:setvalue>`. The first one specifies the value as a literal enclosed in the `<xforms:setvalue>` element. The second possibility uses the `value` attribute: the content of the attribute is an XPath expression evaluated in the context of the node the `xforms:setvalue` element is bound (through the `ref` attribute). The content of the node pointed to by the `ref` attribute will be set with the result of the XPath expression provided in the `value` attribute. The example below and uses two `<xforms:setvalue>`, each one providing the new value in a different way.


```
<xforms: trigger xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms: label>Submit</xforms: label>
  <xforms: action ev:event="DOMActivate">
    <xforms: setvalue ref="clicked">my-button</xforms: setvalue>
    <xforms: setvalue ref="flavor" value="concat('van', 'illa')"/>
  </xforms: action>
</xforms: trigger>
```

The XForms message action displays a message to the user. OPS being a server-side XForms implementation.

Typically, the content of the `message` element is the message to render. It can also come from the binding attributes (`ref` or `bind`), or from the linking attribute (`src`). The order of preference is the following:

- Binding attributes
- Linking attribute
- Inline text



Note

- The only value currently supported for the `level` attribute is `modal`. This attribute is optional.
- When using the linking attribute (`src`), the value must be an absolute URL, starting with `oxf:`, `http:` or other supported protocols.

```
<xforms: trigger>
  <xforms: label>Test</xforms: label>
  <xforms: message ev:event="DOMActivate" ref="taste"/>
</xforms: trigger>
```

Two properties control some aspects of XForms submission in OPS:

```
<property as="xs:boolean" name="oxf.xforms.optimize-post-all" value="true"/>
```

If set to `true` (the default), OPS optimizes submissions with `replace="all"` by sending the response of the submission directly to the web browser. This however means that submission errors cannot be caught by XForms event handlers after OPS has started connecting to the submission URL, as should be the case following XForms 1.0. If set to `false`, OPS buffers the reply so that errors can be handled as per XForms 1.0. However, this solution is less efficient.

```
<property as="xs:boolean" name="oxf.xforms.optimize-local-submission" value="true"/>
```

- If set to `true` (the default), OPS optimizes "local" HTTP and HTTPS submissions, i.e. submissions performed to a URL controlled by OPS itself, by directly submitting using the Java Servlet API instead of actually using the HTTP protocol for the submission.
- If set to `false`, OPS always uses the HTTP or HTTPS protocol, which is less efficient. In this case, it is possible to specify the `xxforms:post` method instead of the `post` method on the `xforms:submission` element to force an optimized local submission.

2.1.5. Formatting

It is usually recommended to use native XML types within XForms instances, as this guarantees interoperability and maintainability. For example, a date of January 10, 2005 is stored in ISO format as: `2005-10-01`. However it is often necessary to format such values on screen in a user-readable format, like "January 10, 2005", "10 janvier 2005", or "10. Januar 2005".

OPS provides an extension attribute, `xxforms:format`, for that purpose. `xxforms:format` must contain an XPath 2.0 expression. In your XPath expression you can use all the XPath 2.0 functions, including those for date manipulation ([external documentation](#)). However since XPath 2.0 functions don't provide any facility for date and time formatting, you can in this attribute also use the following XSLT 2.0 functions:

- `format-date()` ([external documentation](#))
- `format-dateTime()` ([external documentation](#))
- `format-time()` ([external documentation](#))
- `format-number()` ([external documentation](#))

The XPath expression is evaluated by the XForms engine whenever the value bound to the `xforms:input` control changes and needs to be updated on screen. It is evaluated in the context of the instance node bound to the control. This means that the current value of the control can be accessed with `..`. Often the value must be converted, for example to a date, in which case the conversion can be done with XPath 2.0 type casts such as `xs:date(..)`.

When using `xforms:input` and a bound `xs:date` type, you can control the formatting of the date using the `xxforms:format` extension attribute on the `xforms:input` control. For example:

```
<xforms:input ref="date" xxforms:format="format-date(xs:date(..), '[MNN] [D], [Y]', 'en', (), ())"/>
```

When using `xforms:output`, you can control the formatting of the date using the `xxforms:format` extension attribute on the `xforms:input` control.

```
<xforms:output ref="date" xxforms:format="format-date(xs:date(.), '[MNn] [D], [Y]', 'en', (), ())"/>
<xforms:output ref="size" xxforms:format="format-number(., '###,##0')"/>
```

For both `xforms:input` and `xforms:output`, if the bound node is of type `xs:date`, `xs:dateTime` or `xs:time`, and if no `xxforms:format` attribute is present on the control, formatting is based on [properties](#). If the properties are missing, a built-in default formatting is used. The default properties, as well as the built-in defaults, are as follows:

```
<property as="xs:string" name="oxf.xforms.format.date" value="if (. castable as xs:date) then format-date(xs:date
(.), '[FNn] [MNn] [D], [Y]', 'en', (), ()) else ."/>
<property as="xs:string" name="oxf.xforms.format.dateTime" value="if (. castable as xs:dateTime) then format-
dateTime(xs:dateTime(.), '[FNn] [MNn] [D], [Y] [H01]:[m01]:[s01] UTC', 'en', (), ()) else ."/>
<property as="xs:string" name="oxf.xforms.format.time" value="if (. castable as xs:time) then format-time(xs:time
(.), '[H01]:[m01]:[s01] UTC', 'en', (), ()) else ."/>
```

They produce results as follows:

- 2004-01-07 is displayed as Wednesday January 7, 2004
- 2004-01-07T04:38:35.123 is displayed as Wednesday January 7, 2004 04:38:35 UTC
- 04:38:35.123 is displayed as 04:38:35 UTC

Note that with the condition in the XPath expressions, a value which cannot be converted to the appropriate type is simply displayed as is.

2.1.6. XForms Instance Initialization

An XForms page often needs to contain initial data when first loaded. The data may come from a database, a form submitted on a previous page, etc. There are several ways to achieve this with OPS.

Within your page flow, you define a page model and either a static page view:

```
<page id="..." path-info="..." model="my-page-model.xpl" view="my-page-view.xhtml"/>
```

Or a dynamic XSLT page view:

```
<page id="..." path-info="..." model="my-page-model.xpl" view="my-page-view.xsl"/>
```

The page model is in charge of producing an XML document which is then going to be used by the page view to initialize the XForms instance. As always with OPS, the page model produces this document on its data output, and the page view can access this document on its data input, as shown in the following sections. This mechanism is described in details in the [PFC documentation](#).

Following the MVC architecture, the PFC page model generates an XML document which contains an XForms instance. A static PFC page view then includes this instance using `xi:include`, as follows:

```
<html>
  <head>
    <title>Summary</title>
    <xforms:model>
      <xforms:instance id="document-infos-instance">
        <!-- This is where the XML document produced by the page model is included -->
        <xi:include href="input:data"/>
      </xforms:instance>
      ...
    </xforms:model>
  </head>
  <body>
    ...
  </body>
</html>
```

The use of the URI `input:data` instructs XInclude processing to dynamically include the data output of the page view, which is produced on the data output of the page model. Note that it is possible to use the `instance` input, which then refers to the current XML submission.

It is also possible to use a dynamic XSLT page view to perform the inclusion of the instance. XSLT is more flexible than XInclude, but less efficient at runtime. This is an example:

```
<html xsl:version="2.0">
  <head>
    <title>Summary</title>
    <xforms:model>
```

```

<xforms:instance id="document-infos-instance">
  <!-- This is where the XML document produced by the page model is included -->
  <xsl:copy-of select="doc('input:data')/*"/>
</xforms:instance>
...
</xforms:model>
</head>
<body>
...
</body>
</html>

```

Note the use of `xsl:version="2.0"` on the root element of the document, which instructs the PFC to process the page view as an XSLT stylesheet.

The use of the XPath `doc()` function with a URI `input:data` instructs XSLT processing to dynamically include the `data` output of the page view, which is produced on the `data` output of the page model.

Note

It is possible to use XInclude instructions in a dynamic XSLT page view as well. In this case, it is important to note that XInclude instructions are processed before XSLT instructions, i.e. the result of XInclude instructions is an XSLT stylesheet, which is then executed.

Note

Using XSLT for page views has an impact for debugging, as the output of XSLT transformations do not contain valuable location information. For performance and ease of debugging reasons, we recommend using static XHTML views with XInclude whenever possible.

2.1.7. Relative Paths

XForms documents can refer to external resources using URIs in the following circumstances:

- **External Instances.** The `xforms:instance` element can have an `src` attribute linking to an external instance definition.
- **Submission.** The `xforms:submission` element must refer to an action URI.
- **Load Action.** The `xforms:load` action must refer to an URI that must be loaded upon execution.
- **Image Mediatype.** The `xforms:output` control may refer to an image URI.
- **Message, Label, Help, Hint, and Alert.** `xforms:label`, `xforms:help`, `xforms:hint`, and `xforms:alert` may use an `src` attribute to refer to external content.

Note

The XForms 1.1 draft of November 15, 2004 removes linking attributes from actions and metadata elements and "the `src` attribute is not available to XForms 1.1 message, label, help, hint, alert elements."

URIs are resolved relatively to a *base URI*. The base URI is, by default, the external URL used to display the XForms page, with special handling of the servlet context, if necessary. It is also possible to override this behavior by adding `xml:base` attributes on `xforms:load` or any of its ancestor elements.

Referring to external XForms instances is done with the `src` attribute on the `xforms:instance` element:

```
<xforms:instance src="instance.xml" />
```

This feature allows for improved modularity by separating an XForms instance definition from an XHTML page. It also allows for producing XForms instances dynamically.

The following assumes that OPS runs in the `/ops` servlet context:

Base URI (External URL or <code>xml:base</code> attributes)	Initial URI (<code>src</code> attribute)	Resolved URI	Comment
The following URI is loaded in a servlet: <code>http://a.org/ops/page</code>	<code>http://b.com/instance</code>	<code>http://b.com/instance</code>	Absolute URLs are left untouched.
	<code>/new-instance</code>	<code>http://a.org/ops/new-instance</code>	Absolute paths resolve against the current servlet context.
	<code>admin/instance</code>	<code>http://a.org/ops/admin/instance</code>	The relative path resolves against the original URL.

The following path is loaded in a portlet: /example/page	http://b.com/instance	http://b.com/instance	Absolute URLs are left untouched.
	/new-instance	/new-instance	The absolute path is used as is. The XForms instance is loaded from the portlet. The developer must make sure that the path resolves to a PFC entry producing XML.
	admin/instance	/example/admin/instance	The relative path is resolved against the original path. The XForms instance is loaded from the portlet. The developer must make sure that the path resolves to a PFC entry producing XML.

Specifying a submission URL is done with the `action` attribute on the `xforms:submission` element:

```
<xforms:submission action="/submission" ref="..." />
```

The following assumes that OPS runs in the `/ops` servlet context:

Base URI (External URL or <code>xml:base</code> attributes)	Initial URI (action attribute)	XForms Init ¹	Resolved URI	Comment
The following URI is loaded in a servlet: http://a.org/ops/page	http://b.com/submission	N/A	http://b.com/submission	The absolute URL is left untouched. The XForms submission is performed on the absolute URL.
	/new-submission	N/A	http://a.org/ops/new-submission	Absolute paths resolve against the current servlet context.
	admin/submission	N/A	http://a.org/ops/admin/submission	The relative path resolves against the original URL.
The following path is loaded in a portlet: /example/page	http://b.com/submission	N/A	http://b.com/submission	The absolute URL is left untouched. The XForms submission is performed on the absolute URL.
	/new-submission	Yes	/new-submission	The absolute path is used as is. The XForms submission is performed on the portlet.
		No	http://a.org/ops/new-submission	The absolute path resolves against the current servlet context. The submission is performed on the web application.
	admin/submission	Yes	/example/admin/submission	The relative path is resolved against the original path. The XForms submission is performed on the portlet.
		No	http://a.org/ops/example/admin/submission	The relative path resolves against the original path, then against the the current servlet context. The submission is performed on the web application.

¹ If "yes", this means the submission is performed during XForms initialization, for example upon an `xforms-ready` event. If "no", this means that the submission is performed after XForms initialization, for example upon the user activating a trigger.

The `xforms:load` action can refer to a resource to load either through the `resource` attribute or using a single-node binding retrieving the URI from an XForms instance. In both cases, the value of the URI is resolved relatively to the base URI.

The following assumes that OPS runs in the `/ops` servlet context:

Base URI (External URL or <code>xml:base</code> attributes)	Initial URI (resource or Single-Node Binding)	show	Resolved URI	Comment
The following URI is loaded in a servlet: http://a.org/ops/page	http://b.com/software/	replace	http://b.com/software/	The absolute URL is left untouched. The new page replaces the existing page.
		new		The absolute URL is left untouched. A new window or tab opens for the new page.
	/new-page	replace	http://a.org/ops/new-page	Absolute paths resolve against the current servlet context. The new page replaces the existing page.
		new		Absolute paths resolve against the current servlet context. A new window or tab opens for the new page.
	admin/main-page	replace	http://a.org/ops/admin/main-page	The new page replaces the existing page.
		new		A new window or tab opens for the new page.

The following path is loaded in a portlet: /example/page	http://b.com/ software/	replace	http://b.com/ software/	This causes the application to load a page outside of the portlet, replacing the entire portal.
		new		This causes the application to load a page in a new window outside of the portlet.
	/new-page	replace	/new-page	The resulting path is loaded within the portlet.
		new		
	admin/main-page	replace	/example/admin/ main-page	The resulting path is loaded within the portlet.
		new		

When an `xforms:output` control refers to an image URI, as documented [below](#), the resulting value is resolved relatively to the base URI.

2.1.8. XForms and Services

XForms 1.0 allows an XForms page to perform submissions of XForms instances and to handle a response. In most cases, both the submitted XForms instance and the response are XML documents.

Note

It is possible to submit an XForms instance with the HTTP GET method. In that case, some information contained in the XML document is lost, as the structure of the instance, attributes, and namespace prefixes among others, are not passed to the submission.

The XForms submission feature practically allows forms to call XML services. Those services are accessible through an XML API, which means that a request is performed by sending an XML document to the service, and a response consists of an XML document as well.

2.1.9. Extensions

In XForms 1.0, `xforms:output` is used to display text. However, based on a proposal in a draft version of XForms 1.1, OPS supports a `mediatype` attribute on that element.

For the `<xforms:output>` control to display an image, you need to:

- Have a `mediatype` attribute on the `<xforms:output>`. That attribute must refer to an image, such as `image/*` or `image/jpeg`.
- Use the `value` attribute on `<xforms:output>` or bind to the control to a node without type or with an `xs:anyURI` type.

The resulting value is interpreted a URI pointing to an image. The image will display in place of the `xforms:output`. When a single-node binding is used, it is possible to dynamically change the image pointed to. For example:

```
<xforms:output mediatype="image/*" value="' /images/moon.jpg' " />
```

```
<xforms:model>
  <xforms:instance>
    <image-uri />
  </xforms:instance>
  <xforms:bind nodeset="image-uri" type="xs:anyURI" />
</xforms:model>
...
<xforms:output mediatype="image/*" ref="image-uri" />
```

Note

It is not yet possible to directly embed image data in an XForms instance using the `xs:base64Binary` type.

When an `xforms:output` control has a `mediatype` attribute with value `text/html`, the value of the node to which the control is bound is interpreted as HTML content. Consider the following XForms instance:

```
<xforms:instance id="my-instance">
  <form>
    <html-content>
      This is in &lt;t; b> &gt; bol d&lt;t; /b> &gt; !
    </html-content>
  </form>
</xforms:instance>
```

You bind an `xforms:output` control to the `html-content` node as follows:

```
<xforms:output ref="instance('my-instance')/html-content" mediatype="text/html" />
```

This will display the result as HTML, as expected: "This is in in **bold**". If the `mediatype` is not specified, the result would be instead: "This is in in bold". In the XForms instance, the HTML content must be escaped as text. On the other hand, the following content will not work as expected:

```
<xforms:instance>
  <form>
    <html-content>
      This is in in<b>bold</b>!
    </html-content>
  </form>
</xforms:instance>
```

Note

When using a `mediatype="text/html"`, an HTML `<div>` element will be generated by the XForms engine to hold the HTML data. As in HTML a `<div>` cannot be embedded into a `<p>`, if you have a `<xforms:output mediatype="text/html">` control, you should not put that control into a `<xhtml:p>`.

OPS implements some extension functions which can be used from XPath expressions in XForms documents.

When using XPath 2.0, the following functions from XSLT 2.0 are also available:

- `format-date()` ([external documentation](#))
- `format-dateTime()` ([external documentation](#))
- `format-time()` ([external documentation](#))
- `format-number()` ([external documentation](#))

The following functions are implemented:

- `xxforms:call-xpl($xplURL as xs:string, $inputNames as xs:string*, $inputElements as element()*, $outputNames as xs:string+)`.

This function lets you call an XPL pipeline.

1. The first argument, `$XPLurl`, is the URL of the pipeline. It must be an absolute URL.
2. The second argument, `$inputNames`, is a sequence of strings, each one representing the name of an input of the pipeline that you want to connect.
3. The third argument, `$inputElements`, is a sequence of elements to be used as input for the pipeline. The `$inputNames` and `$inputElements` sequences must have the same length. For each element in `$inputElements`, a document is created and connected to an input of the pipeline. Elements are matched to input name by position, for instance the element at position 3 of `$inputElements` is connected to the input with the name specified at position 3 in `$inputNames`.
4. The fourth argument, `$outputNames`, is a sequence of output names to read.
5. The function returns a sequence of elements corresponding the output of the pipeline. The returned sequence will have the same length as `$outputNames` and will correspond to the pipeline output with the name specified on `$outputNames` based on position.

The example below shows a call to the `xxforms:call-xpl` function:

```
xxforms:call-xpl ('oxf:/examples/sandbox/xpath/run-xpath.xpl', ('input', 'xpath'), (instance('instance')/input, instance('instance')/xpath), 'formatted-output')/*, 'html')
```

eXForms is a suggested set of extensions to XForms 1.0, grouped into different modules. OPS supports the `exf:mip` module, which includes the following functions:

- `exf:relevant()`
- `exf:readonly()`
- `exf:required()`

eXForms functions live in the `http://www.exforms.org/exf/1-0` namespace, usually bound to the prefix `exf`.

2.1.10. State Handling

The OPS XForms engine requires keeping processing state while operating on an XForms page. Such state includes the current values of XForms instances, selected repeated elements, and more. With OPS, XForms state information can be handled in one of two ways:

- **Client-side:** in this case, static initial state information is sent along with the initial HTML page, and dynamic state is exchanged over the wire between the client browser and the OPS XForms server when necessary.

Benefits of the approach:

- The OPS server is entirely stateless. It only requires memory while processing a client request. It can be restarted without consequence for the XForms engine.
- State information does not expire as long as the user keeps the application page open in the web browser.

Drawbacks of the approach:

- Resulting HTML pages are larger. In particular, the size of state data grows when XForms instances grow, regardless of whether many XForms controls are bound to instance data.
- More data circulates between the client browser and the OPS XForms server.

Note

OPS compresses and encrypts XForms state information sent to the client.

- **Server-side:** in this case, state information is stored on the server, in association with an application session. Only very little state information circulates between client and server.

Benefits of the approach:

- Resulting HTML page are smaller. HTML pages increase in size as more XForms controls are used, but they don't increase in size proportionally to the size of XForms instances.
- Small amounts of data circulate between the client browser and the OPS XForms server.
- This means that very large XForms instances can be processed without any impact on the amount of data that is transmitted between the client and the server.

Drawbacks of the approach:

- The OPS XForms server needs to be stateful. It uses server memory to store state information in a session even when no request is being processed. The server must be configured to determine how much state information is kept in a session, how long session take to expire, etc. This creates additional demand for resources on the server and complicates the task of tuning the server.
- State information can become unavailable when sessions expire or when the server is restarted (unless you setup the server to persist session information). When state information becomes unavailable for a page, that page will no longer function unless it is reloaded.

Note

With most servlet containers, it is possible to configure session handling to passivate sessions out of the application server memory to a persistent store. It is this way possible to partially alleviate the drawback above by making sure that a very large number of active but idle sessions can be kept, with a minimum impact on application server memory. It is this way also possible to make sure that sessions survive a servlet container restart.

Note

OPS ensures that it is possible to open multiple client browser windows showing the same page within the same session.

State handling can be configured globally for all pages, or locally for each individual page served. Global configuration is performed in `properties.xml` with the `oxf.xforms.state-handling` property. When missing or set to `client`, state is stored client-side. When set to `session`, state is stored server-side in a session. For example:

```
<!-- Store state in the session -->
<property as="xs:string" name="oxf.xforms.state-handling" value="session"/>
```

The global configuration can be overridden for each page by setting the `xxforms:state-handling` attribute in the page. This attribute can be set for example on the root element of the XHTML page, or on the first `xforms:model` element. Only the first such attribute encountered by the XForms engine is used:

```
<xforms:model xxforms:state-handling="client">...</xforms:model>
```

When storing state in a session, the maximum size of the data to be stored for each user can be selected using the `oxf.xforms.cache.session.size` property. The size is specified in bytes:

```
<!-- Allow a maximum of 500 KB of state information for each user -->
<property as="xs:integer" name="oxf.xforms.cache.session.size" value="500000"/>
```

Whether state information is kept client-side or server-side, a property controls whether the XForms engine should try to optimize state reconstruction by using a cache. This property should usually be set to `true`:

```
<!-- This should usually be set to "true" -->
<property as="xs:boolean" name="oxf.xforms.cache.document" value="true"/>
```

While XForms gets you a long way towards creating a dynamic user-friendly user interface, there are some dynamic behaviors of the user interface that cannot be implemented with XForms, and that you might want to implement directly in JavaScript. We describe here how your JavaScript code can interact with XForms and in particular how you can have access to the value of an XForms control.

In JavaScript, you obtain a reference to the element representing the control with `var control = document.getElementById('myControl')` where `myControl` is the id of the XForms control (i.e. `<xforms:input id="myControl">`). You can read the current value of the control with `control.value` and set the value of the control assigning a value to `control.value`, for instance: `control.value = 42`. When you do such an assignment the value of the node in the instance is updated, the `xforms-value-changed` event thrown, and all the standard XForms processing happens (validation, recalculation, etc).

Note

Currently, accessing the value of a control is only supported for `<xforms:input>`. If you want to access the value of another control, a workaround consists in adding an `<xforms:input>` with `style="display: none"` bound to the same node as that control, and accessing the `<xforms:input>`'s id to change the value of the node and therefore indirectly the other control's value.

2.2. Page Flow Controller

2.2.1. Introduction to the Page Flow Controller

The OPS Page Flow Controller (PFC) is the heart of your OPS web application. It dispatches incoming user requests to individual pages built out of models and views, following the model / view / controller (MVC) architecture.

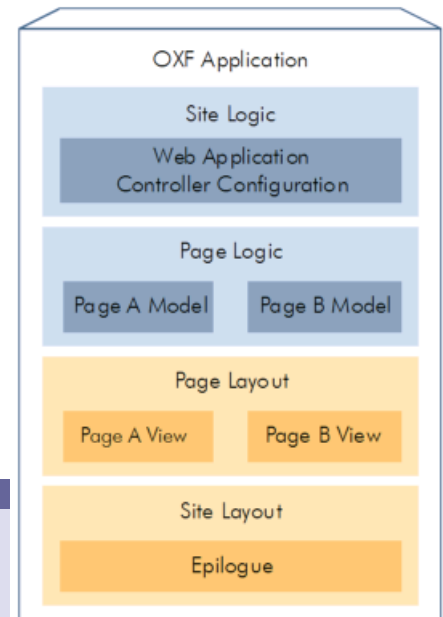
The PFC is configured with a file called a *page flow*. A page flow not only describes the pages offered by your application, it also declaratively describes an entire application's navigation logic, allowing the development of pages and XML services completely independently from each other.

The PFC encourages designing applications with a total separation between:

- **Site Logic or Page Flow**: when, and how to navigate from one page to another.
- **Page Logic** (the MVC page model): how data entered by the user is processed (for example validated, then fed to a backend), and how data is retrieved from a backend.
- **Page Layout** (the MVC page view): how information is displayed and presented to the user.
- **Site Presentation**: the layout and look and feel common to all pages in the web application or the web site, e.g.: site navigation menus, headers and footers, table backgrounds, or number formatting.

Note

By default, the PFC is configured in `web.xml` as the main processor for the OPS servlet and portlet. However, you are not required to use the PFC with OPS: you can define your own main processor for servlets and portlets, as documented in [Packaging and Deployment](#). For most web applications, the PFC should be used.



2.2.2. Basics

A page flow is usually defined in a file called `page-flow.xml` stored at the root of your OPS resources. This XML file has a root element called `<config>`, which has to be within the `http://www.orbeon.com/oxf/controller` namespace. All the XML elements in a page flow have to be in that namespace unless stated otherwise. You start a page flow document as follows:

```
<config>...</config>
```

You can configure the location of the page flow configuration file in the web application's `web.xml` file. See [Packaging and Deployment](#) for more information. In most cases, it is not necessary to change the default name and location.

Most web applications consist of a set of *pages*, identified to clients (web browsers) by a certain URL, for example:

```
http://www.orbeon.org/myapp/report/detail?first=12&count=10#middle
```

In most cases the URL can be split as follows:

- `http://www.orbeon.org/` identifies the web or application server hosting the application.
- `/myapp` may optionally identify the particular web application running on that server. Whilst this part of the URL path is not mandatory, its use is encouraged on Java application servers, where it is called the *context path*.
- `/report/detail` identifies the particular page within the web application. Such a path may be "flat", or hierarchical, separated with "/" (slashes).
- `?first=12&count=10` specifies an optional *query string* which identifies zero or more *parameters* passed to that page. Each parameter has a value. This example has two parameters: the first one is called `first` with value 12, and the second one is called `count` with value 10.

- #middle is an optional fragment identifier identifying a section of the resulting page. Usually, this part of the URL is not handled by the web application, instead the web browser uses it to scroll to a section of the resulting page identified by this identifier (here `middle`).

For a particular web application, what usually matters in order to identify a particular page is the path within the URL under the context path, here `/report/detail`. Therefore, in an OPS page flow, each page is identified with a unique path information. You declare a minimal page like this:

```
<page path-info="/report/detail" />
```

Other pages may be declared as follows:

```
<page path-info="/report/summary" />
<page path-info="/home" />
```

A `<page>` element can have an optional `id` attribute useful for navigating between pages.

Creating a static page with OPS is quite easy: just add a `view` attribute on a `<page>` element which points to an XHTML file:

```
<page path-info="/report/detail" view="oxf:/report/detail/report-detail-view.xhtml" />
```

Here, using the `oxf:` protocol means that the file is searched through the OPS resource manager sandbox. It is also possible to use relative paths, for example:

```
<page path-info="/report/detail" view="report/detail/report-detail-view.xhtml" />
```

The path is relative to the location of the page flow configuration file where the `<page>` element is contained. Here is an example of the content of `report-detail-view.xhtml`:

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

It is recommended to use XHTML and to put all the elements in the XHTML namespace, `http://www.w3.org/1999/xhtml`. This can be done by using default namespace declaration on the root element (`xmlns="http://www.w3.org/1999/xhtml"`) or by mapping the namespace to a prefix such as `xhtml` and to use that prefix throughout the document, as shown above. The file must contain well-formed XML: just using a legacy HTML file won't work without some adjustments, usually minor.

Instead of using a static XHTML page, you can also use an XSLT template to generate a dynamic page. This allows using XSLT constructs mixed with XHTML constructs, for example:

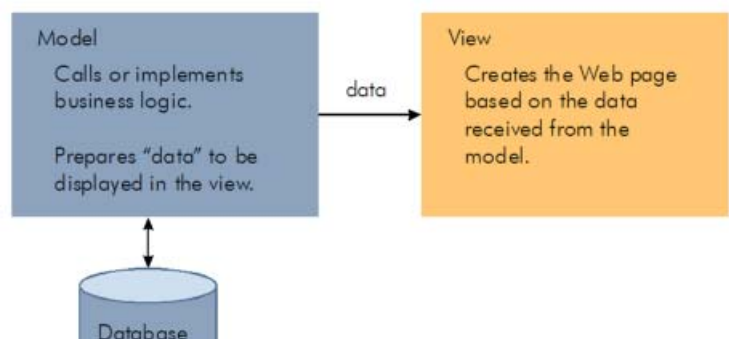
```
<html xsl:version="2.0">
  <head>
    <title>Current Time</title>
  </head>
  <body>
    <p>
      The time is now<xsl:value-of select="current-dateTime()" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
    </p>
  </body>
</html>
```

When XSLT templates are used, it is recommended to use the `.xsl` extension:

```
<page path-info="/report/detail" view="report/detail/report-detail-view.xsl" />
```

In the MVC architecture, the page logic is implemented by a *page model*, and the page layout by a *page view*. The MVC architecture promotes the separation of model, view and controller:

- **The page model** is responsible for calling or implementing the business logic. It is in charge of gathering the information to be displayed by the page view.
- **The page view** is in charge of presenting to the user the information gathered by the page model. The page view usually produces XHTML and XForms, but it can also produce other results such as XSL-FO, RSS, etc. Handling of the output of the



view is done in the [page flow epilogue](#), which by default knows how to handle XHTML, XForms, XSL-FO, and custom XML document.

- **The controller** is responsible for dispatching a request from a client such as a web browser to the appropriate page model and view and connecting the model with the view. In OPS, the controller is the PFC itself, which is configured with a page flow.

For instance, a news page can use a page model to retrieve the list of headlines and then pass this information as an XML document to a page view. The view produces an XHTML page by creating a table with the content of the headlines, adding a logo at the top of the page, a copyright notice at the bottom, etc.

Each PFC `<page>` element therefore supports attributes defining what page model and page view must be used:

- The `model` attribute is a URL referring to an [XPL pipeline](#) (optionally an XSLT stylesheet or a static XML file) implementing the model.
- The `view` attribute is a URL referring to an XSLT stylesheet (optionally an [XPL pipeline](#) or a static XML file) implementing the view.

The model passes data to the view as an XML document, as follows:

- **XPL model.** The model document must be generated by the [XPL pipeline](#) on an output named `data`.
- **XSLT model.** The model document is the default output of the XSLT transformation.
- **Static XML model.** The model document is the static XML document specified.
- **XPL view.** The model document is available on an input named `data`.
- **XSLT view.** The model document is available as the default input of the XSLT transformation.
- **Static XML view.** In this case, no model document is available to the view.

A model [XPL pipeline](#) and an XSLT view can be declared as follows for the `/report/detail` page:

```
<page path-info="/report/detail" model="report/detail/report-detail-model.xpl" view="report/detail/report-detail-view.xsl"/>
```

Here, the location of the model and view definitions mirrors the path of the page, and file names repeat the directory path, so that files can be searched easier. It is up to the developer to choose a naming convention, but it is recommended to follow a consistent naming structure. Other possibilities include:

```
<page path-info="/report/detail" model="report-detail-model.xpl" view="report-detail-view.xsl"/>
```

or:

```
<page path-info="/report/detail" model="models/report-detail-model.xpl" view="views/report-detail-view.xsl"/>
```

A typical XSLT view can extract model data passed to it automatically by the PFC on its default input, for example, if the model generates a document containing the following:

```
<employee>
  <name>John Smith</name>
</employee>
```

Then an XSLT view can display the content of the `<name>` element as follows:

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

2.2.3. XML Submission

A page built out of a model and a view can retrieve information from a data source and format it. However, this is not enough to make a page which can use parameters submitted by a client to configure what data is being presented, or how it is presented.

The OPS PFC uses the concept of *XML submission* to provide page configurability. To the model and view of a given page, an XML submission is simply an XML document whose content is available as an [XPL pipeline](#) or an XSLT input called `instance`.

There are different ways to produce an XML submission:

- **Internal XForms submission.** The built-in OPS XForms engine uses an HTTP POST XForms submission to submit an XForms instance.
- **External submission.** An external application or a client-side XForms engine uses HTTP POST to submit an XML document directly to a page.

- **PFC page navigation.** The PFC, based on a user configuration, produces an XML document to submit internally to a given page.
- **Default submission.** Each page can refer to a *default submission document* containing an XML document automatically submitted to the page if no other submission is done.

The most common case of XML submission in OPS is submission from the built-in XForms engine. Assume you have a page defined as follows:

```
<page path-info="/report/detail" model="report/detail/report-detail-model.xpl" view="report/detail/report-detail-view.xml"/>
```

If you wish to submit an XForms instance to this page from within `report-detail-view.xml`, create an XForms submission as follows:

```
<xforms:submission id="main" method="post" action="/report/detail" xmlns:xforms="http://www.w3.org/2002/xforms"/>
```

This ensures that when this XForms submission is activated, an XML document containing the submitted XForms instance will be made available to the page model and view.

Note

The `action` attribute on the `xforms:submission` element should not be confused with the `<action>` element of the page flow. The former specifies a URL to which the XForms submission must be performed, as per the XForms 1.0 recommendation; the latter specifies a PFC *action* executed when a specified boolean XPath expression operating on an XML submission evaluates to true. The XForms submission's `action` attribute instead matches a PFC `<page>` element's `path-info` attribute.

You can also directly submit to another page by specifying a different action, for example:

```
<xforms:submission id="main" method="post" action="/report/summary" xmlns:xforms="http://www.w3.org/2002/xforms"/>
```

In general it is recommended to leave the control of the flow between pages to PFC *actions*, as documented below.

An external XML submission must refer to the URL of the page accepting the submission. It is up to the developer to provide this URL to the external application, for example `http://www.orbeon.org/myapp/xmlrpc` if you have a page declaring the path `/xmlrpc`:

```
<page path-info="/xmlrpc" model="xmlrpc.xpl"/>
```

The *Blog* and *Employees* examples illustrate the implementation of XML-RPC and web services handlers using this mechanism.

In case there is no external or internal XML submission, it is possible to specify a static default XML submission document. This is particularly useful to extract information from a page URL, as documented below. You specify a default submission with the `default-submission` attribute as follows:

```
<page path-info="/report/detail" default-submission="report-detail-default-submission.xml"/>
```

The mechanisms described above explain how a page receives an XML submission, but not how to actually access the submitted XML document. You do this in one of the following ways:

- **XPL model.** The model accesses the XML submission document from its `instance` input.
- **XSLT model.** The model accesses the XML submission document using the `doc('input:instance')` function.
- **Static XML model.** The model cannot access the XML submission document.
- **XPL view.** The view accesses the XML submission document from its `instance` input.
- **XSLT view.** The view accesses the XML submission document using the `doc('input:instance')` function.
- **Static XML view.** The view cannot access the XML submission document.

If no submission has taken place, the XML submission document is an OPS "null" document, as follows:

```
<null xsi:nil="true"/>
```

XML submission using HTTP POST convenient in many cases, however there are other ways page developers would like to configure the way a page behaves:

- **Using URL parameters.** URL parameters are specified in a query string after a question mark in the URL, explained above.
- **Using URL path elements.** URL paths can be hierarchical, and the elements of the paths can have a user-defined meaning.

A PFC page can easily extract data from the URL using the `<setvalue>` element nested within the `<page>` element. To do so, an XML submission must take place on the page. This can be achieved by using the default submission if no other submission is taking place. The default submission document must contain placeholders for the values to extract from the URL. Given an URL query string of `first=12&count=10` with two parameters, `first` and `count`, a default submission document can look as follows:

```
<submission>
  <first/>
  <count/>
</submission>
```

The following page extracts the two URL parameters:

```
<page path-info="/report/detail" default-t-submission="report-detail-default-t-submission.xml">
  <setvalue ref="/submission/first" parameter="first"/>
  <setvalue ref="/submission/count" parameter="count"/>
</page>
```

The `<setvalue>` element uses the `ref` attribute, which contains an XPath 2.0 expression identifying exactly one element or attribute in the XML document. The text value of the element or attribute is then set with the value of the URL parameter specified. If there is no parameter with a matching name, the element or attribute is not modified. This allows using default values, for example:

```
<submission>
  <first/>
  <count>5</count>
</submission>
```

In such a case, if no `count` parameter is specified on the URL, the default value will be available.

With a query string of `first=12&count=10`, the resulting XML document will be:

```
<submission>
  <first>12</first>
  <count>10</count>
</submission>
```

With a query string of `first=12`, the resulting XML document will be:

```
<submission>
  <first>12</first>
  <count>5</count>
</submission>
```

Note

The default submission document does not have to use element or attribute names identical to the URL parameter names. Doing so however may make the code clearer.

If there are multiple URL parameters with the same name, they will be stored in the element or attribute separated by spaces.

It is also possible to extract data from the URL path. To do so, use a `matcher` attribute on the page as [documented below](#). You can then extract regular expression groups using the `<setvalue>` element with the `matcher-group` attribute:

```
<page path-info="/blog/([^/]+)/([^/]+)" matcher="oxf:perl5-matcher" default-t-submission="recent-posts/recent-posts-
default-t-submission.xml" model="recent-posts/recent-posts-model.xpl" view="recent-posts/recent-posts-view.xpl">
  <setvalue ref="/form/username" matcher-group="1"/>
  <setvalue ref="/form/blog-id" matcher-group="2"/>
</page>
```

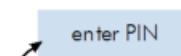
The `matcher-group` attribute contains a positive integer identifying the number of the regular expression group to extract. With a path of `/blog/jdoe/456`, the first group contains the value `jdoe`, and the second group the value `456`.

Note

If a page actually uses an XML submission, which means either having `<action>` elements, or reading the instance in the page model or page view, it must not expect to be able to read the HTTP request body separately using the [Request generator](#).

2.2.4. Navigating Between Pages

The site logic or page flow describes the conditions that trigger the navigation from one page to the other. It also describes how arguments are passed from one page to the other. In a simple web



application simulating an ATM, as illustrated by the [ATM example](#) the navigation logic could look like the one described in the diagram on the right. In this diagram, the square boxes represent pages and diamond-shaped boxes represent actions performed by the end-user.

With the PFC, page flow is expressed declaratively and externally to the pages. Consequently, pages can be designed independently from each other. The benefits of a clear separation between site logic and page logic and layout include:

- **Simplicity:** the site logic is declared in one place and in a declarative way. You don't need to write custom logic to perform redirects between pages or pass arguments from page to page.
- **Maintainability:** having different developers implementing independent page is much easier. Since the relationship between pages is clearly stated in the page flow, it also becomes much easier to modify a page in an existing application without affecting other pages.



Consider a `view-account` page in the ATM web application. The page displays the current balance and lets the user enter an amount of money to withdraw from the account. The page looks like this:

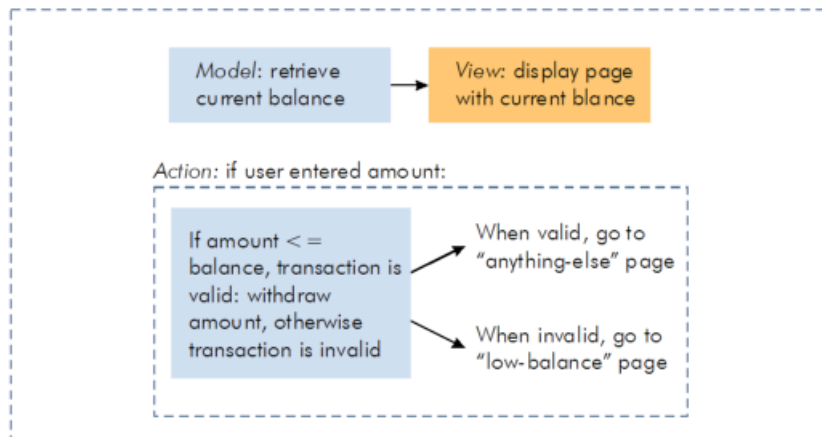
ATM Example: View Account

Balance: 100

This page is composed of different parts illustrated in the figure below:

- **The page model** retrieves the current balance.
- **The page view** displays the balance, and presents a form for the user to enter the amount to withdraw.
- **An action** executed when the user enters an amount in the text field. This action checks if the amount entered is inferior or equal to the account balance. If it is, the balance is decreased by the amount entered and the transaction is considered valid. Otherwise, the transaction is considered illegal. Depending on the validity of the transaction, a different page is displayed. If the transaction is valid, the `anything-else` page is displayed; otherwise the `low-balance` page is displayed.

Page: for path-info = `"/view-account"`



This behavior is described in the Page Flow with:

```

<page id="view-account" path-info="/view-account" model="view-account-get-balance-model.xpl" view="view-account-get-balance-view.xsl">
  <action when="/amount != ''" action="view-account-action.xpl">
    <result id="success" when="/success = 'true'" page="anything-else"/>
    <result id="failure" when="/success = 'false'" page="low-balance"/>
  </action>
</page>

```

On the `<page>` element, as documented above:

- The `path-info` attribute tells the PFC what relative URL corresponds to this page. The URL is relative to the application context path.
- The `model` attribute points to the page model XPL pipeline.
- The `view` attribute points to the page view XSLT template.

The `<page>` element contains an `<action>` element. It is named *action* because it is typically executed as a consequence of an action performed by the end-user, for example by clicking on a button or a link which causes a form to be submitted. There may be more than one `<action>` element within a `<page>` element. On an `<action>` element:

- The `when` attribute contains an XPath 2.0 expression executed against the XML submission. The first `<action>` element with a `when` attribute evaluating to `true()` is executed. The `when` attribute is optional: a missing `when` attribute is equivalent to `when="true()"`. Only the last `<action>` element is allowed to have a missing `when` attribute. This allows for defining a default action which executes if no other action can execute.
- When the action is executed, if the optional `action` attribute is present, the [XPL pipeline](#) it points to is executed.

The `<action>` element can contain zero or more `<result>` elements.

- If an `action` attribute is specified on the `<action>` element, the `<result>` element can have a `when` attribute. The `when` attribute contains an XPath 2.0 expression executed against the data output of the action [XPL pipeline](#). The first `<result>` evaluating to `true()` is executed. The `when` attribute is optional: a missing `when` attribute is equivalent to `when="true()"`. Only the last `<result>` element is allowed to have a missing `when` attribute. This allows for defining a default action result which executes if no other action result can execute.
- A `<result>` element optionally has a `page` attribute. The `page` attribute contains a page id pointing to a page declared in the same page flow. When the result is executed and the `page` attribute is present, the destination page is executed, and the user is forwarded to the corresponding page.

Note

In this case, a page model or page view specified on the enclosing `<page>` element does not execute! Instead, control is transferred to the page with the identifier specified. This also means that the [page flow epilogue](#) does not execute.

- A `<result>` element can optionally contain an XML submission. The submission can be created using XSLT, XQuery, or the deprecated XUpdate. You specify which language to use with the `transform` attribute on the `<result>` element. The inline content of the `<result>` contains then a transformation in the language specified. Using XSLT is recommended.

The transformation has automatically access to:

- An `instance` input, containing the current XML submission. From XSLT, XQuery and XUpdate, this input is available with the `doc('input:instance')` function. If there is no current XML submission, a "null" document is available instead:

```
<null xsi:nil="true"/>
```

- An `action` input, containing the result of the action [XPL pipeline](#) if present. From XSLT, XQuery and XUpdate, this input is available with the `doc('input:action')` function. If there is no action result, a "null" document is available instead:

```
<null xsi:nil="true"/>
```

- The default usually contains the current XML submission as available from the `instance` input.

For backward compatibility however, when the destination page has an `xforms` attribute, the default input contains instead the destination page's resulting XForms instance.

The result of the transformation is automatically submitted to the destination page. If there is no destination page, it replaces the current XML submission document made available to the page model and page view.

An action [XPL pipeline](#) supports an optional `instance` input containing the current XML submission, and produces an optional data output with an action result document which may be used by a `<result>` element's `when` attribute, as well as by an XML submission-producing transformation. This is an example of action [XPL pipeline](#):

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:param name="instance" type="input"/>
  <p:param name="data" type="output"/>
  <!-- Call the data access layer -->
  <p:processor name="oxf:pipeline">
    <p:input name="config" href="../../data-access/delegate/read-document.xml"/>
    <p:input name="document-id" href="#instance#xpather(/document-id)/>
    <p:output name="document-info" ref="data"/>
  </p:processor>
</p:config>
```

Notice the `instance` input, the `data` output, as well as a call to a data access layer which uses information from the XML submission and directly returns an action result document.

The following is an example of using XSLT within `<result>` element in order to produce an XML submission passed to a destination page:

```
<action when="/form/action = 'show-detail'" action="../../bizdoc/summary/find-document-action.xml">
  <result page="detail" transform="oxf:xslt">
    <form xsl:version="2.0">
      <document-id>
        <xsl:value-of select="doc('input:action')/document-info/document-id" xmlns:xsl="http://
www.w3.org/1999/XSL/Transform"/>
      </document-id>
    </form>
  </result>
</action>
```

```

</document-i d>
<document>
  <xsl:copy-of select="doc('input:action')/document-info/document/*" xmlns:xsl="http://www.w3.org/
    1999/XSL/Transform"/>
</document>
</form>
</result>
</action>

```

Notice the transform attribute set to `oxf:xslt`, and the use of the `doc('input:action')` to refer to the output of the action XPL pipeline specified by the action attribute on the `<action>` element. The current XML submission can also be accessed with `doc('input:instance')`.

See the following examples to understand how `<page>`, `<action>`, and `<result>` can be used in a page flow:

- The simple [ATM](#) example.
- The [BizDoc](#) example.

Also see the [OPS Tutorial](#).

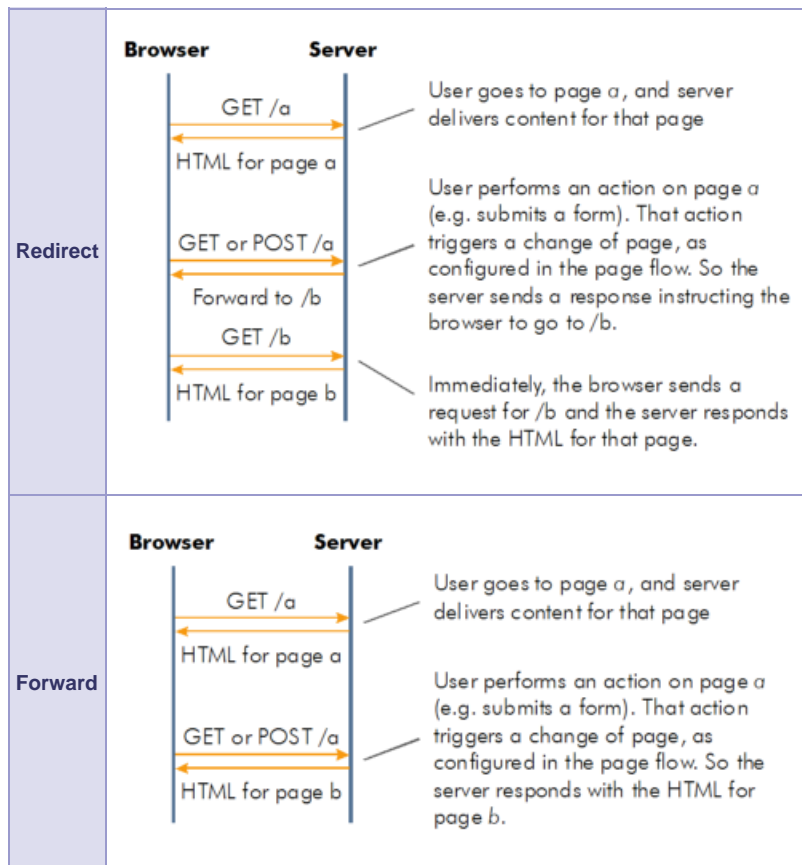
You can control what method is used to perform an internal XML submission described within a `<result>` element. Consider this page flow:

```

<page id="a" path-info="/a" model="..." view="...">
  <action when="...">
    <result page="b"/>
  </action>
</page>
<page id="b" path-info="/b" model="..." view="...">

```

Going from page "a" to page "b" can be done with either a "forward" or a "redirect" method:



The benefit of the "redirect" method is that after being redirected to page *b*, the end-user will see a URL starting with `/b` in the browser's address bar. He will also be able to bookmark that page and to come back to it later. However, a drawback is that the request for page *b* is sent by the browser with a `GET` method. Since browsers impose limits on the maximum amount of information that can be sent in a `GET` (URL length), this method might not work if the amount of information that needs to be passed to page *b* from page *a* is too large. This typically happens when working with fairly large XML submissions. In those cases, you must use the "forward" method, which does not limit the amount of information passed from page to page. The "forward" method also reduces the number of roundtrips with the server.

Note

A third instance passing option, `redirect-exit-portal`, behaves like the `redirect` method but sends a redirection which exits the portal, if any. This is mainly useful for the OPS examples portal.

2. At the page flow level with the `instance-passing` attribute on the page flow root element:

```
<config instance-passing="forward|redirect">...</config>
```

3. In the page flow at the "result" level, with the `instance-passing` attribute on the `<result>` element:

```
<page id="a" path-info="/a" model="..." view="...">
  <action when="...">
    <result page="b" instance-passing="forward|redirect"/>
  </action>
</page>
```

A configuration at the application level (`properties.xml`) can be overridden by a configuration at the page flow level (`instance-passing` on the root element), which can in its turn be overridden by a configuration at the result level (`instance-passing` on the `<result>` element).

2.2.5. Other Configuration Elements

A page flow file is comprised of three sections:

- The `<files>` elements list files that must be served directly to the client, such as images or CSS files.
- The `<page>` elements declare pages and for each one specify identifier, path, model, view, and XML submission.
- The `<epilogue>` and `<not-found-handler>` elements define additional behavior that apply to all the pages.

Some files are not dynamically generated and are served to the client as-is. This is typically the case for images such as GIF, JPEG, CSS files, etc..

You tell the PFC what files to serve directly with one or more `<files>` elements in the page flow. The example below shows the configuration used by the OPS examples:

```
<config>
  <files path-info="*.gif"/>
  <files path-info="*.css"/>
  <files path-info="*.pdf"/>
  <files path-info="*.js"/>
  <files path-info="*.png"/>
  <files path-info="*.jpg"/>
  <files path-info="*.wsdl"/>
  <files path-info="*.html" mime-type="text/html"/>
  <files path-info="*.java" mime-type="text/plain"/>
  <files path-info="*.txt" mime-type="text/plain"/>
  <files path-info="*.xq" mime-type="text/plain"/>
  ...
</config>
```

With `<files path-info="*.gif"/>`, if a request reaches the PFC with the path `images/logo.gif`, the file `oxf:/images/logo.gif` is sent in response to that request.

The `<files>` element supports the `path-info` and `matcher` attributes like the `<page>` element. It also supports a `mime-type` attribute telling the PFC what media type must be sent to the client with the files. The PFC uses defaults for well-known extension, as defined by the [Resource Server processor](#). In doubt, you can specify the `mime-type` attribute.

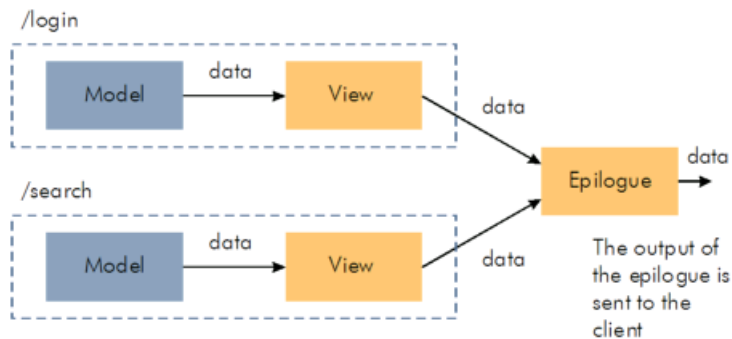
The value of the `path-info` attribute can be either a simple pattern or a custom pattern.

Value	Description
Simple Pattern	Simple patterns optionally start or end with a star character (*). For instance: <code>/about/company.html</code> matches exactly this URL, <code>about/*</code> matches any URL that starts with <code>about/</code> , <code>*.gif</code> matches any URL that ends with <code>.gif</code> .
Custom Pattern	In this case, an additional <code>matcher</code> attribute must be specified on the <code><page></code> element. The <code>matcher</code> argument points to a matcher processor qualified name. Two matcher processors are provided with OPS: the Perl5 matcher (<code>oxf:perl5-matcher</code>) and the Glob matcher (<code>oxf:glob-matcher</code>). The Perl5 matcher accepts Perl regular expressions and the Glob matcher accepts Unix shell glob expressions .

This is an example of `<files>` element using the Perl5 matcher:

```
<files path-info="/doc/[^.]*\.html" matcher="oxf:perl5-matcher"/>
```

As with the `<files>` element, a matcher can also be specified on a `<page>` element. When using a matcher that allows for groups, the part of the path matched by those groups can be extracted as documented above with the `<setvalue>` element. Note that the only matcher bundled with OPS that accepts groups is the Perl5 matcher.



You often want a common look and feel across pages. Instead of duplicating the code implementing this look and feel in every page view, you can define it in a central location called the *page flow epilogue*. The `<epilogue>` element specifies the [XPL pipeline](#) which implements the page flow epilogue.

This is an example of `<epilogue>` element, pointing to the default epilogue XPL pipeline:

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

The page flow epilogue is discussed in more details in the [Page Flow Epilogue](#) documentation.

The `<not-found-handler>` element is used to specify the identifier of a page to call when no `<page>` element in the page flow is matched by the current request. There can be only one `<not-found-handler>` per page flow.

This is an example of `<not-found-handler>` element and the associated `<page>` element:

```
<!-- "Not Found" page displayed when no page matches the request URL -->
<page id="not-found" path-info="/not-found" view="/config/not-found.xml"/>
<not-found-handler page="not-found"/>
```

By default, `oxf:/config/not-found.xml` displays a simple XHTML page telling the user that the page requested cannot be found.

Note

The `<not-found-handler>` element is not used for resources served through the `<files>` element. In that case, the PFC returns instead a "not found" code to the user agent (code 404 in the case of HTTP).

2.2.6. Error Handling

Several things can go wrong during the execution of a page flow by the PFC, in particular:

- The page flow may be ill-formed.
- A runtime error may be encountered when processing the page flow, such as not finding a particular page model referenced by a page.
- An action, page model, page view or epilogue may generate an error at runtime.

Those error conditions are not directly handled by the PFC. Instead, they are handled with the error [XPL pipeline](#) specified in the web application's `web.xml` file. By default, the error processor is the Pipeline processor, which runs the `oxf:/config/error.xpl` XPL pipeline. You can configure `error.xpl` for your own needs. By default, it formats and displays the Java exception which caused their error.

See [Packaging and Deployment](#) for more information about configuring error processors.

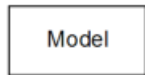
2.2.7. Typical Combinations of Page Model and Page View

The sections below show how page model and page view are often combined.

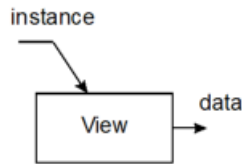
Simple pages with no back-end code can be implemented with a single [XPL pipeline](#), XSLT template or static page. A view XPL pipeline must have a `data` output. The XML generated by the view then goes to the epilogue.



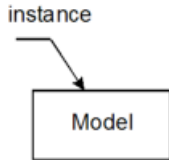
If a page is not sent back to the user agent, there is no need for a view. This is typically the case when a redirect needs to be issued, a binary file is produced, or when a page simply implements an XML service.



This is a variant of the *view only* scenario, where an XML submission is used. In this case, the view receives the XML submission as the *instance* input.



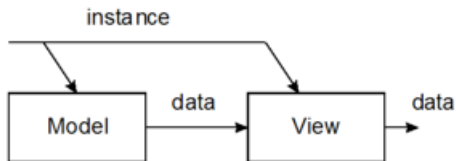
This is a variant of the *model only* scenario, where an XML submission is used.



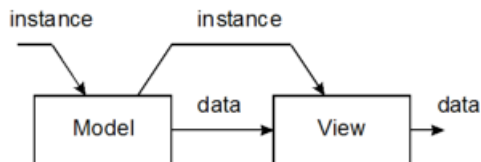
This is the classic case. An XPL pipeline implements the page model and an XSLT template implements the page view where data produced by the model is consumed by the view.



This is the equivalent of the previous model where an XML submission is used. In this case an *instance* input is made available to the model and the view.



This is a variant of the previous case where the model declares an *instance* output. This allows the model to modify the submitted XML instance. This is typically useful when the view displays some values from the XML submission document but these values are not exactly the same as those entered by the user. For example, a page with a text field where the user types an airport code. If the user enters a known city such as San Francisco, the application may automatically add the corresponding airport code (SFO in this case).



2.2.8. Examples

The following example illustrates how to perform a simple redirection with the PFC. Assume you want some path, */a*, to be redirect to another path, */b*. You can do this as follows:

```

<page path-info="/a">
  <action>
    <result page="page-b" instance-passing="redirect"/>
  </action>
</page>
<page id="page-b" path-info="/b">...</page>
  
```

Note that you do not have to use *redirect*, but that doing so will cause the user agent to display the path to page */b* in its URL bar. The *instance-passing* attribute is also unnecessary if *redirect* is already the default instance passing mode.

The PFC allows you to very easily receive an XML document submitted, for example with an HTTP POST, and to generate an XML response. This can be useful to implement XML services such as XML-RPC, SOAP, or any XML-over-HTTP service. The following PFC configuration defines a simple XML service:

```
<page path-info="/xml rpc" model="xml-rpc.xpl" />
```

Notice that there is no `view` attribute: all the processing for this page is done in the page model.

The following content for `xml-rpc.xpl` implements an XML service returning as a response the POST-ed XML document:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <!-- The XML submission is available on the "instance" input -->
  <p:param name="instance" type="input" schema-href="request.rng"/>
  <!-- Processing of the XML submission (here we just return the request) -->
  <p:processor name="oxf:identity">
    <p:input name="data" href="#instance"/>
    <p:output name="data" id="response"/>
  </p:processor>
  <!-- Generate a response -->
  <p:processor name="oxf:xml-serializer">
    <p:input name="data" href="#response" schema-href="response.rng"/>
    <p:input name="config">
      <config/>
    </p:input>
  </p:processor>
</p:config>
```

Notice the optional `schema-href` attributes which allow validating the request and the response against schemas.

2.3. Page Flow Epilogue

2.3.1. Introduction to the Page Flow Epilogue

The OPS *page flow epilogue* is an *XPL pipeline* executed by the *Page Flow Controller* (PFC) in order to perform tasks common to all the pages in an OPS application. Such tasks include:

- **Look and feel.** Adding a common look and feel to all the XHTML pages in an application.
- **Page layout.** Adding a page header, footer, or navigation bar.
- **Linking common files.** Adding common links to external CSS stylesheets or scripts.
- **Document-aware formatting.** Formatting the document differently depending on the type of document produced by a page view, for example XHTML or XSL-FO.
- **Client-aware formatting.** Producing XHTML or plain HTML depending on the user agent requesting the page, or different versions of HTML depending on different clients.
- **Server-side XForms processing.** The OPS server-side XForms engine is hooked up in the epilogue.
- **URL rewriting.** Rewriting URLs so that XHTML pages are written independently from their final location.
- **Servlet and portlet support.** Formatting pages differently for deployment as a portlet.
- **User-defined configuration.** Finally, because the epilogue is a configurable XPL pipeline, there is no limit to the tasks it can accomplish!

Thanks to the page flow epilogue, it is not necessary to duplicate common behavior in all the page views of a given OPS application.

2.3.2. Basics

The epilogue pipeline is specified with the `<epilogue>` element in your page flow. By default, the *standard epilogue* located under `oxf:/config/epilogue.xpl` is used.

The epilogue is executed by the PFC only after a page view executes. This means that for a given execution of a page, the following conditions must be met for the epilogue to execute:

- The `<page>` elements has a `view` attribute.
- No page result element with a `page` attribute executes (because in that case, a redirection to another page occurs, and neither model nor view of the current page executes).

Type	Name	Purpose
		When executed, the epilogue XPL pipeline has a <code>data</code> input containing the XML document produced by the page view which just executed. In particular, if the page view is: <ul style="list-style-type: none">■ A static XML file: the <code>data</code> input directly receives the unmodified content of that file.

		<ul style="list-style-type: none"> ■ An XSLT template: the data input receives the default output of the XSLT template. ■ An XPL pipeline: the data input receives the content of the data output of that pipeline.
Input	instance	The epilogue XPL pipeline receives on its <code>instance</code> input the XML submission performed on the page, if any. The epilogue usually does not use this input.
Input	xforms-model	The epilogue XPL pipeline also receives on its <code>xforms-model</code> input the legacy XForms model as specified on the <code><page></code> element's <code>xforms</code> attribute. This input is for the private use of the legacy server-side XForms engine in OPS. You usually do not have to deal with it.
Output	-	The epilogue XPL pipeline does not have any output: the result of the XPL pipeline must be produced using a serializer, usually the HTTP serializer , which serializes the resulting data directly to an HTTP client such as a web browser.

2.3.3. The Standard Epilogue: `epilogue.xpl`

The standard epilogue is found under `oxf:/config/epilogue.xpl`. It performs the following tasks:

- **XForms processing:** it applies server-side XForms processing if needed. If your application does not use the OPS built-in XForms engine at all, you can bypass XForms processing.
- **Container type selection:** it dispatches to two sub-epilogues, `oxf:/config/epilogue-servlet.xpl` and `oxf:/config/epilogue-portlet.xpl`, depending on whether the page is running in a servlet or portlet environment. If your application does not use portlets at all, you can bypass this choice.

Note

With OPS 3.0, only the servlet environment is officially supported. The portlet environment is however used for the examples portal. Please also refer to this [FAQ entry](#).

This means that for your OPS applications, `epilogue-servlet.xpl` is usually the relevant file to consider.

2.3.4. The Servlet Epilogue: `epilogue-servlet.xpl`

`oxf:/config/epilogue-servlet.xpl` is the epilogue file you are the most likely to configure.

Type	Name	Purpose
Input	xformed-data	This XPL pipeline receives on this input the XML document produced by the page view and then transformed by server-side XForms processing. This means that if the page view produces an XHTML + XForms document, the document received by <code>epilogue-servlet.xpl</code> on its <code>xformed-data</code> is already transformed by the server-sided XForms engine into XHTML.
Input	data	This XPL pipeline also receives on its <code>data</code> input the raw XML document produced by the page view. This is the same XML document received by <code>epilogue.xpl</code> .
Output	-	This XPL pipeline does not have any output: the result of the XPL pipeline must be produced using a serializer, usually the HTTP serializer , which serializes the resulting data directly to an HTTP client such as a web browser.

`epilogue-servlet.xpl` performs different tasks depending on the input XML document.

- **XHTML:** for XHTML documents, identified with a root element of `xhtml:html` (in the XHTML namespace), two options are proposed:
 - **Option 1: conversion from XHTML to HTML.**
 1. The standard theme is applied.
 2. URLs are rewritten.
 3. XHTML is converted to HTML by moving the XHTML elements to no namespace and by removing their prefixes, if any.
 4. The result is converted to plain HTML according to the [XSLT 2.0 and XQuery 1.0 Serialization](#) HTML output method.
 5. The resulting HTML document is sent to the client through HTTP.
 - **Option 2: browser detection with native XHTML output option.** This option is commented out so that the default is just to send plain HTML to the client.
 1. The standard theme is applied.

2. URLs are rewritten.
3. If the client tells the server, with an `accept` header, that it accepts XHTML documents:
 1. XHTML elements are stripped from their prefixes, if any, but they are left in the XHTML namespace. This makes the XHTML more compatible with certain non-XHTML-aware user agents.
 2. The result is converted to serialized XHTML according to the [XSLT 2.0 and XQuery 1.0 Serialization](#) XHTML output method.
4. Otherwise:
 1. XHTML is converted to HTML by moving the XHTML elements to no namespace and by removing their prefixes, if any.
 2. The result is converted to plain HTML according to the [XSLT 2.0 and XQuery 1.0 Serialization](#) HTML output method.
5. The resulting document is sent to the client through HTTP.

Note

You can easily construct variations on those options, such as sending XHTML only to certain user agents, or always sending XHTML.

- **Pseudo-HTML:** for pseudo-HTML documents, identified with a root element of `html` (in lowercase and in no namespace):

1. URLs are rewritten.
2. The result is converted to plain HTML according to the [XSLT 2.0 and XQuery 1.0 Serialization](#) HTML output method.
3. The resulting HTML document is sent to the client through HTTP.

Note

No theme stylesheet is applied in this case. You are encouraged to write XHTML and use a single theme which matches against XHTML elements only, leaving possible conversion to HTML for subsequent steps. It is however possible to apply a theme stylesheet in this case as well by inserting a call to the XSLT processor in this case as well.

It should also be noted that URL rewriting only applies to lowercase HTML elements. URL rewriting for pseudo-HTML documents is deprecated.

- **XSL-FO:** if the document is an XSL-FO document, identified with a root element of `fo:root` (in the XSL-FO namespace):

1. The XSL-FO converter is called. It generates a binary PDF document.
2. The resulting PDF document is sent to the client through HTTP.

- **Binary and text documents:** if the document root is `document` and has an `xml:type` attribute:

1. The encapsulated binary or text document is sent as per the semantic described in [Non-XML Documents](#).

- **Plain XML:** in all other cases:

1. The result is converted to plain XML according to the [XSLT 2.0 and XQuery 1.0 Serialization](#) XML output method.
2. The plain XML document is sent to the client through HTTP.

You are free to:

- Modify the support for the document formats handled by default.
- Add support for his own document formats by adding `p:when` branches to the `p:choose` statement.

2.3.5. The Portlet Epilogue: `epilogue-portlet.xpl`

`oxf:/config/epilogue-portlet.xpl` is the epilogue file used for the examples portal. You usually do not have to worry about this XPL pipeline unless you add new examples, or unless you deploy as a JSR-168 portlet.

`epilogue-portlet.xpl` has the same inputs as the `epilogue-servlet.xpl` epilogue.

`epilogue-portlet.xpl` performs different tasks depending on the input XML document.

- **XHTML:** for XHTML documents, identified with a root element of `xhtml:html` (in the XHTML namespace):

1. The standard theme is applied but an XHTML fragment enclosed in a `xhtml:div` element is produced instead of a full-fledged XHTML document.
2. URLs are rewritten.
3. The resulting XHTML document is sent to the portlet container as an XHTML fragment.

- **Plain XML:** in all other cases:

1. The XML document is included in a simple XHTML document and formatted into XHTML.

2. The resulting XHTML document then goes through the same steps as a regular XHTML document..

You are free to:

- Modify the support for the document formats handled by default.
- Add support for his own document formats by adding `p:when` branches to the `p:choose` statement. In a portlet environment, it is important to remember that a portlet can usually only output a markup fragment such as HTML or XML, but no associated resources such as images, PDF files, etc.

2.3.6. XForms Processing

The OPS built-in [XForms engine](#) allows you to write not only raw XHTML, but also supports XForms 1.0. To achieve this, the XForms engine rewrites XHTML + XForms into XHTML + Javascript + CSS. This transformation also occurs in the epilogue, as mentioned [above](#).

While it is not necessary to know in details the work performed by the XForms engine, it is important to know that:

- XForms processing is triggered when either a legacy XForms model is found on the `xforms-model` input of the epilogue, or when an XForms model is found under `/xhtml:html/xhtml:head`.
- The result of XForms processing is that all the elements in the XForms namespace are removed, including the XForms models you may have under `xhtml:head`.
- Instead of XForms elements, you will find XHTML elements. Those elements may use some CSS classes defined in `xforms.css`.
- The standard theme includes some default OPS CSS stylesheets and scripts used by the XForms engine.

2.3.7. The Standard Theme

The standard OPS theme is found under the `oxf:/config/theme` directory. The entry point of the theme is the `theme.xsl` stylesheet, called from the epilogue XPL pipelines.

As documented above, the standard theme is called only for views that generate XHTML documents. The standard theme does not run, for example, on pseudo-HTML, XSL-FO, or plain XML documents.

The standard theme does the following:

- It copies, under `xhtml:head`, the `xhtml:meta`, `xhtml:link`, `xhtml:style`, and `xhtml:script` elements from the source XHTML document. This allows you for example to link to a particular Javascript script on a page by page basis, instead of including the script in all the pages of your application.
- It adds, under `xhtml:head`, links to OPS CSS stylesheets and scripts necessary for the server-side XForms engine, in particular the following CSS stylesheets:
 - **orbeon.css**: definitions proprietary to the default theme. These definitions are likely to be overridden by the user as they create their own theme.
 - **xforms.css**: definitions related to XForms controls. These definitions may be changed by users if they are not happy with the defaults.
- It generates, under `xhtml:head`, an `xhtml:title` element, first by looking in the XHTML document if present. If not found, it looks for a `f:example-header` element (this element is used by some OPS examples). Finally, it looks for the first `xhtml:h1` element in the body of the XHTML document.
- It handles the `f:tabs` element used by some OPS examples.
- It processes the source `xhtml:body` element. This mostly results in copying the content of the source `xhtml:body`, with the following exceptions:
 - Styling is applied to some XHTML forms elements.
 - Attribute in the XHTML namespace are copied but in no namespace.
 - Some special handling is performed for the body of included portlets so that URLs do not get rewritten in this case. This is useful only for the examples portal.
 - Any element within the `http://orbeon.org/oxf/xml/formatting` namespace, usually prefixed with `f:`, is handled by the included `formatting.xsl` stylesheet. This stylesheet contains some useful formatting templates, in particular to format XML and display tabs. None of the templates in this stylesheet is required for your OPS application.

The categories above are implemented with sets of `xsl:template` elements. Those are clearly delimited in `theme.xsl`.

You can configure the standard theme at your leisure for your own application. You typically get started by modifying `theme.xsl`.

2.4. Introduction to the XML Pipeline Definition Language (XPL)

2.4.1. Introduction

The XML Pipeline Definition Language (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.

XPL pipelines are built up from smaller components called XML processors or XML components. 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 OPS comes standard with a comprehensive library. XPL orchestrates these to create business logic, similar to the way Java code "orchestrates" method calls within a Java object.

For a step-by-step introduction to pipelines, see the [OPS Tutorial](#).

Please also refer to the [XPL 1.0 Submission at W3C](#).

2.4.2. XPL Interpreter

The XPL interpreter is itself implemented as an XML processor, called the Pipeline processor. This processor reads a pipeline definition following the XPL syntax on its `config` input, and assembles a pipeline according to that definition. It is then able to run the pipeline when called.

2.4.3. Namespace

All the elements defined by XPL must be in the namespace with a URI: <http://www.orbeon.com/oxf/pipeline>. For consistency, XPL elements should use the `p` prefix. This document we will assume that this prefix is used.

2.4.4. <p:config> element

The root element of a XPL document (`config`) defines:

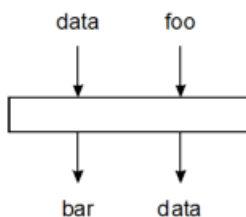
- Zero or more input or output parameters to the pipeline with `<p:param>`
- The list of statements that need to be executed for this pipeline. A statement defines either a processor with its connections to other processors in the pipeline using `<p:processor>`, or a condition using `<p:choose>`.

The `<p:config>` element and its content are defined in the Relax NG schema with:

```
<start>
  <ref name="config"/>
</start>
<define name="config">
  <element name="p:config">
    <optional>
      <attribute name="id"/>
    </optional>
    <ref name="param"/>
    <ref name="statement"/>
  </element>
</define>
<define name="statement">
  <interleave>
    <zeroOrMore>
      <ref name="processor"/>
    </zeroOrMore>
    <zeroOrMore>
      <ref name="choose"/>
    </zeroOrMore>
    <zeroOrMore>
      <ref name="for-each"/>
    </zeroOrMore>
  </interleave>
</define>
```

2.4.5. <p:param> element

The `<p:param>` element defines what the inputs and outputs of the pipeline are. Each input and output has a name. There cannot be two inputs with the same name or two outputs with the same name, but it is possible to have an output and an input with the same name. Every input name defines an id that can be later referenced with the `href` attribute such as when connecting processors. The output names can be referenced with the `ref` attribute on `<p:output>`.



The inputs and outputs of the above pipeline are declared in the XPL document below:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:param type="input" name="data"/>
  <p:param type="input" name="foo"/>
  <p:param type="output" name="bar"/>
  <p:param type="output" name="data"/>
</p:config>
```

The `<p:param>` element and its content are defined in the Relax NG schema with:

```

<define name="param">
  <zeroOrMore>
    <element name="p: param">
      <interleave>
        <attribute name="name"/>
        <attribute name="type"/>
      </interleave>
    </element>
  </zeroOrMore>
</define>

```

2.4.6. <p:processor> element

The <p:processor> element places a processor in the pipeline and connects it to other processors, pipeline inputs, or pipeline outputs.

- The kind of processor created is specified with the `name` attribute, which is an XML qualified name. A qualified name is composed of two parts:
 - **A prefix:** The prefix is mapped to a URI defining a namespace.
 - **A local name:** This name is a name in the namespace defined by the prefix.

This mechanism allows grouping related processors in a namespace. For example, all the basic OPS processors are grouped in the `http://www.orbeon.com/oxf/processors` namespace. This namespace is typically mapped to the `oxf` prefix. Processors are then referred to using names such as `oxf:xslt` or `oxf:scope-serializer`.

The name maps to a processor factory. Processor factories are registered through the `processors.xml` file described in [Packaging and Deployment](#).

Note

For backward compatibility, the `uri` attribute is still supported.

- The <p:input> element connects the input of the processor with the name specified with the `name` attribute to one of:
 - an inline XML document embedded in the <p:input> element
 - an XML document obtained according to the [full syntax of the href attribute](#), for example:
 - `href="#some-id"`
 - `href="oxf:/my-document.xml"`
 - `href="aggregate('document', #some-id, oxf:/my-document.xml#xpointer(/*/*))"`
- The <p:output> element defines an `id` corresponding to that output with the `id` attribute or connects the output to a pipeline output with the `ref` attribute.
- Optionally, <p:input> and <p:output> can have a `schema-href` or `schema-uri` attribute. Those attributes specify a schema that is used by the Pipeline processor to validate the corresponding input or output. `schema-href` references a document using the [href syntax](#). `schema-uri` specifies the URI of a schema that is mapped to a specific schema in the [OPS properties file](#).
- Optionally, <p:input> and <p:output> can have a `debug` attribute. When this attribute is present, the document that passes through that input or output is logged with Log4J. This is useful during development to watch XML documents going through the pipeline.

The following example feeds an XSLT processor with an inline document and an external stylesheet.

```

<p:processor name="oxf:xslt" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config" href="stylesheet.xsl"/>
  <p:input name="data" schema-href="oxf:/address-book-schema.xml">
    <address-book>
      <card>
        <name>John Smith</name>
        <email>j@example.com</email>
      </card>
      <card>
        <name>Fred Bloggs</name>
        <email>fb@example.net</email>
      </card>
    </address-book>
  </p:input>
  <p:output name="data" id="address-book"/>
</p:processor>

```

The <p:processor> element and its content are defined in the Relax NG schema with:

```

<define name="processor">
  <element name="p:processor">
    <attribute name="name"/>
    <interleave>
      <zeroOrMore>
        <element name="p:input">
          <attribute name="name"/>
          <ref name="debug"/>
          <ref name="schemas"/>
          <optional>

```



```

        <choi ce>
            <attri bute name="href" />
            <ref name="anyEl ement" />
        </choi ce>
    </opti onal >
</el ement>
</zeroOrMore>
<zeroOrMore>
    <el ement name="p: output">
        <attri bute name="name" />
        <ref name="schemas" />
        <ref name="debug" />
        <choi ce>
            <attri bute name="id" />
            <attri bute name="ref" />
        </choi ce>
    </el ement>
</zeroOrMore>
</i nterl eave>
</el ement>
</defi ne>

```

2.4.7. <p:choose> element

The <p:choose> element can be used to execute different processors depending on a specific condition. The general syntax for this is very close to XSLT:

```

<p: choose href="#condi ti on-document" xmlns:p="http://www.orbeon.com/oxf/pi pel i ne">
    <p: when test="fi rst-condi ti on">... </p: when>
    <p: when test="second-condi ti on">... </p: when>
    <p: otherwi se>... </p: otherwi se>
</p: choose>

```

The conditions are expressed in XPath and operate on the XML document specified by the href attribute on p:choose. Each branch can contain regular processor declarations as well as nested conditions.

Outputs declared in a branch are subject to the following conditions:

- An output id cannot override an output id in scope before the corresponding choose element
- The scope of an output id is local to the branch if it is connected inside that branch
- The set of output ids not connected inside a branch become visible to processors declared after the corresponding choose element
- The set of output ids not connected inside the branch must be consistent among all branches

The last condition means that if a branch has two non-connected outputs such as output1 and output2, then all other branches must declare the same outputs. On the other hand, inputs in branches do not have to refer to the same outputs.

The <p:choose> element and its content are defined in the Relax NG schema with:

```

<defi ne name="choose">
    <el ement name="p: choose">
        <attri bute name="href" />
        <oneOrMore>
            <el ement name="p: when">
                <attri bute name="test" />
                <ref name="statement" />
            </el ement>
        </oneOrMore>
        <opti onal >
            <el ement name="p: otherwi se">
                <ref name="statement" />
            </el ement>
        </opti onal >
    </el ement>
</defi ne>

```

2.4.8. <p:for-each> element

With <for-each> you can execute processors multiple times based on the content of a document. Consider this example: an XML document contains information about employees, each described in an emp element. This document is stored in a file called company.xml:

```

<company>
    <emp>
        <fi rstname>John</fi rstname>
        <l astname>Smi th</l astname>
    </emp>
    <emp>
        <fi rstname>Judy</fi rstname>
        <l astname>Mat thews</l astname>
    </emp>

```

```

<emp>
  <firstname>Gloria</firstname>
  <lastname>Schwartz</lastname>
</emp>
</company>

```

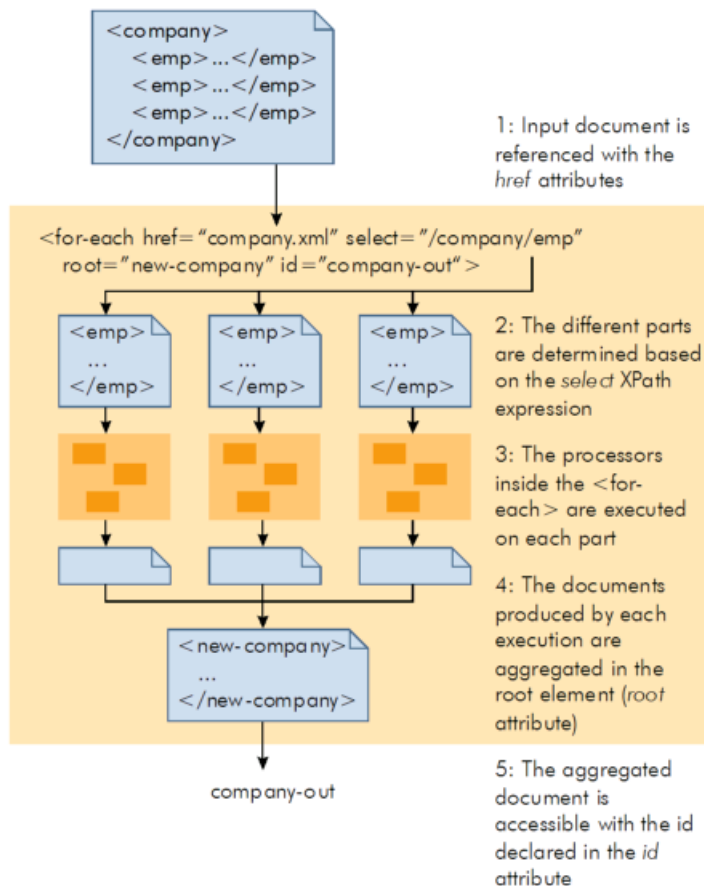
You want to apply a stylesheet (stored in transform-employee.xsl) to each employee. You can do this with the following pipeline:

```

<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:for-each href="company.xml" select="/company/emp" root="new-company" id="company-out">
    <p:processor name="oxf:xslt">
      <p:input name="data" href="current()" />
      <p:input name="config" href="transform-employee.xsl" />
      <p:output name="data" ref="company-out" />
    </p:processor>
  </p:for-each>
  <!-- The id "company-out" can now be referenced by other -->
  <!-- processor in the pipeline. -->
</p:config>

```

This diagram describes how the iteration is done in the above example:



- In a <for-each> you can have multiple processors connected together, <choose> statements and nested <for-each>, just like outside of a <for-each>.
- The output of a processor (or other <for-each>) inside the <for-each> must be "connected to the for-each" using a ref="..." attribute. The value in the ref attribute must match the value of the <for-each> id attribute.
- You access the current part of the XML document being iterated with current() in an href expression. If you have nested <for-each>, current() applies to the <for-each> that directly includes the current() expression.
- The processor inside a <for-each> can access ids declared before the <for-each> statement.
- The aggregated document (the "output of the <for-each>") is available in the rest of the pipeline with the id declared in the id attribute. Alternatively, you can directly connect the output of the <for-each> to an output of the current pipeline with a ref attribute (as in the processor <output> element). If the ref attribute is used (instead of id), then the value of the ref must be referenced (instead of the value of the id attribute). When both the id and ref attributes are used, the value of the id attribute must be referenced.
- The <for-each> can have optional attributes: input-debug, input-schema-href, input-schema-uri, output-debug, output-schema-href and output-schema-uri. The attributes starting with "input" (respectively "output") work as the similar attributes, just without the prefix, on the <input> element (respectively <output> element). The attributes starting with "input" apply to the document referenced by the href expression. The attributes starting with "output" apply to the output of the <for-each>.

2.4.9. href attribute

The `href` attribute is used to:

- Reference external documents
- Refer outputs of other processors
- Aggregate documents using the `aggregate()` function
- Select part of a document using XPointer

The complete syntax of the `href` attribute is described below in a Backus Naur Form (BNF)-like syntax:

```
href          ::= ( local_reference | uri | aggregation ) [ xpointer ]
local_reference ::= "#" id
aggregation   ::= "aggregate(" root_element_name "," agg_parameter ")"
root_element_name ::= "'" name "'"
agg_parameter  ::= href [ "," agg_parameter ]
xpointer      ::= "#xpointer(" xpath_expression ")"
```

The URI syntax is defined in RFC 2396. A URI is used to reference an external document. A URI can be:

- Absolute, if a protocol is specified. For instance `file:/dir/file.xml`.
- Relative, if no protocol is specified. For instance `../file.xml`. The document is loaded relatively to the URL of the XPL document where the `href` is declared, as specified in RFC 1808.

Multiple documents can be aggregated with the `aggregate()` function. The name of the root element that will contain the aggregated document is specified in the first argument. The documents to aggregate are specified in the following arguments. There is no restriction on the number of documents that can be aggregated.

For example, you have a document (with output id `first`):

```
<empl oyee>John</empl oyee>
```

And a second document (with output id `second`):

```
<empl oyee>Marc</empl oyee>
```

Those two documents can be aggregated using `aggregate('employees', #first, #second)`. This produces the following document:

```
<empl oyees>
  <empl oyee>John</empl oyee>
  <empl oyee>Marc</empl oyee>
</empl oyees>
```

The XPointer syntax is used to select parts of a document. For example, if you have a document in a file called `company.xml`:

```
<company>
  <name>Orbeon</name>
  <site>
    <web>http://www.orbeon.com/</web>
    <ftp>ftp://ftp.orbeon.com/</ftp>
  </site>
</company>
```

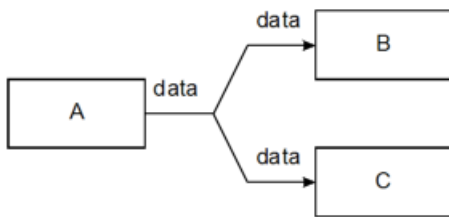
The expression `company.xml#xpointer(/company/site)` produces the document:

```
<site>
  <web>http://www.orbeon.com/</web>
  <ftp>ftp://ftp.orbeon.com/</ftp>
</site>
```

The same id may be referenced multiple times in the same XPL document. For example, the id `doc` is referenced by two processors in the following example:

```
<p: config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: processor uri="A">
    <p: output name="data" id="doc"/>
  </p: processor>
  <p: processor uri="B">
    <p: input name="data" href="#doc"/>
  </p: processor>
  <p: processor uri="C">
    <p: input name="data" href="#doc"/>
  </p: processor>
</p: config>
```

The document seen by B and C are identical. This situation can be graphically represented as:



2.4.10. Processor Inputs and Outputs

XPL processors declare a certain number of inputs and outputs. Those inputs and outputs constitute the *interface* of the processor, in the same way that methods in object-oriented programming languages like Java expose parameters. For example, the XSLT processor expects:

- a `config` input receiving an XSLT stylesheet definition
- a `data` input receiving the XML document to transform
- a `data` output producing the transformed XML document

You know what inputs and outputs to connect for a given processor by consulting the documentation for that processor. This is similar to looking up a method signature in an object-oriented programming language.

Consider the following XSLT processor instance in a pipeline:

```
<p:processor name="oxf:xslt" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config" href="stylesheet.xml"/>
  <p:input name="data" schema-href="oxf:/address-book-schema.xml">
    <address-book>
      <card>
        <name>John Smith</name>
        <email>js@example.com</email>
      </card>
      <card>
        <name>Fred Bloggs</name>
        <email>fb@example.net</email>
      </card>
    </address-book>
  </p:input>
  <p:output name="data" id="address-book"/>
</p:processor>
```

Both its `config` and `data` inputs are said to be connected, because the `<p:processor>` element for the XSLT processor has `<p:input>` elements for both those inputs, and they each refer to an XML document:

- In the first case, a resource called `stylesheet.xml`
- In the second case, an inline document with root element `address-book`

There are other ways to connect inputs, for example:

```
<p:config xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <!-- Pipeline input called "my-input" -->
  <p:param name="my-input" type="input"/>
  <!-- First XSLT transformation -->
  <p:processor name="oxf:xslt">
    <p:input name="config" href="stylesheet-1.xml"/>
    <p:input name="data" href="#my-input"/>
    <p:output name="data" id="address-book"/>
  </p:processor>
  <!-- Second XSLT transformation -->
  <p:processor name="oxf:xslt">
    <p:input name="config" href="stylesheet-2.xml"/>
    <p:input name="data" href="#address-book"/>
    <p:output name="data" id="phone-list"/>
  </p:processor>
  <!-- ... -->
</p:config>
```

In this case:

- The `data` input of the first XSLT processor instance is connected to the `my-input` input of the pipeline.
- The `data` input of the second XSLT processor instance is connected to the `address-book` output of the first XSLT processor instance.

The example above shows that the `address-book` output of the first XSLT processor instance is connected to the input of a following processor. A processor output can also be connected to a pipeline output, as follows:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <!-- Pipeline input called "my-input" -->
  <p:param name="my-input" type="input"/>
  <!-- Pipeline output called "my-output" -->
  <p:param name="my-output" type="output"/>
  <!-- XSLT transformation -->
  <p:processor name="oxf:xslt">
    <p:input name="config" href="stylesheet-1.xsl"/>
    <p:input name="data" href="#my-input"/>
    <p:output name="data" ref="my-output"/>
  </p:processor>
</p:config>
```

In this case, the `data` output of the XSLT processor is connected to the `my-output` output of the containing pipeline.

To sum up, a processor input can be connected to:

- a resource XML document
- an inline XML document
- the output of another processor
- a pipeline input
- a combination of the above through the [full syntax of the href attribute](#)

A processor output can be connected to:

- the input of another processor with the `id` attribute
- a pipeline output with the `ref` attribute

Some inputs and outputs are *required* by a processor. This means that you have to declare `<p:input>` and `<p:output>` elements with the appropriate `name` attribute within the `<p:processor>` element corresponding to that processor, and to connect those inputs and outputs as discussed in the previous section. Most processors require all their inputs and outputs to be connected.

Some processors on the other hand may declare some inputs and outputs as *optional*. This means that the user of the processor may or may not connect an input or output if it is not necessary to do so. For example, the SQL processor declares an optional `datasource` input. If the `datasource` input is needed by the user, it must be connected:

```
<p:processor name="oxf:sql" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="datasource" href="my-datasource.xml"/>
  <p:input name="data" href="#some-data"/>
  <p:input name="config">
    <config>
      ...
    </config>
  </p:input>
</p:processor>
```

On the other hand, if the user of the SQL processor does not require an external `datasource` document, she can simply not connect the `datasource` input:

```
<p:processor name="oxf:sql" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data" href="#some-data"/>
  <p:input name="config">
    <config>
      ...
    </config>
  </p:input>
</p:processor>
```

It is entirely up to each processor to determine which inputs and outputs are mandatory or optional, and how and when they are read.

Note that a processor may decide whether an input must be connected depending on the content of other inputs, for example the SQL processor does not require the `datasource` input if its `config` input already refers to a J2EE `datasource`. On the contrary, if it does not refer to such a `datasource`, it requires the `datasource` input to be connected. If it is not, the processor generates an error at runtime.

In certain cases, the user of a processor must refer, from a processor configuration, to particular processor inputs and outputs. If you implement a new processor, you should support the `input:` and `output:` URI schemes for this purpose. On the other hand, if you are using a standard OPS processor supporting such references to processor inputs and outputs, you can count on the `input:` and `output:` URI schemes being used. For example:

- To refer to a processor input named `my-input`, use the URI: `input:my-input`
- To refer to a processor output named `my-output`, use the URI: `output:my-output`

While there is no requirement for processor configurations to follow this URI convention, it is highly recommended to do so whenever possible to ensure consistency. In OPS, several processors make use of it, including:

- The XSLT processor
- The XUpdate processor
- The Email processor

For concrete examples, please refer to the [XSLT processor](#) or the [Email processor](#) documentation.

Currently, no standard processor within OPS makes uses of the `output:` scheme. The XSLT processor would be a good candidate for this feature, with XSLT 2.0's support for multiple output documents.

3. OPS Technologies Reference

3.1. Resource Managers

3.1.1. Introduction

A *resource manager* is an Orbeon PresentationServer (OPS) component responsible for reading and writing XML and other resources like binary and text documents. A resource manager abstracts the actual mechanisms used to store resources. An important benefit of using such an abstraction is that it is possible to store all your application files in a sandbox which can be moved at will within a filesystem or between storage mechanisms. For instance, resources can be stored:

- As files on disk (using your operating system's file system)
- As resources within a WAR file
- As resources within one or more JAR files
- In a WebDAV server

A resource manager is both used:

- Internally by OPS
- By your own OPS applications through URLs with the `oxf:` protocol

This chapter describes the different types of resource managers and explains their configuration.

3.1.2. General Configuration

A single resource manager is initialized per OPS web application. Configuration is handled in the web application descriptor (`web.xml`) by setting a number of context parameters. The first parameter indicates the resource manager factory:

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-value>
    org.orbeon.oxf.resources.DefaultResourceManagerFactory
  </param-value>
</context-param>
```

Other properties depend on the resource manager defined by the factory. These properties are described in the following sections.

3.1.3. Filesystem Resource Manager

Purpose	Loading resources from a filesystem
Factory	<code>org.orbeon.oxf.resources.FilesystemResourceManagerFactory</code>
Properties	<code>oxf.resources.filesystem.sandbox-directory</code>

The Filesystem resource manager loads resources from a filesystem. This is especially useful during development, since no packaging of resources is necessary. The `oxf.resources.filesystem.sandbox-directory` property can be used to define a sandbox, and if used must point to a valid directory on a filesystem. If not specified, no sandbox is defined, and it is possible to access all the files on the filesystem.

Using the Filesystem resource manager without a sandbox is particularly useful for command-line applications.

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-value>
    org.orbeon.oxf.resources.FilesystemResourceManagerFactory
  </param-value>
</context-param>
<context-param>
  <param-name>oxf.resources.filesystem.sandbox-directory</param-name>
  <param-value>/home/user/oxf/myapp/resources</param-value>
</context-param>
```

3.1.4. ClassLoader Resource Manager

Purpose	Loading resources from a JAR file in the classpath
Factory	<code>org.orbeon.oxf.resources.ClassLoaderResourceManagerFactory</code>
Properties	None

The class loader resource manager can load resource from a JAR file or from a directory in the classpath. This resource manager is required to load internal resources for OPS.

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-value>
    org.orbeon.oxf.resources.ClassLoaderResourceManagerFactory
  </param-value>
</context-param>
```

3.1.5. WebApp Resource Manager

Purpose	Loading resources from a WAR file or deployed Web Application
Factory	org.orbeon.oxf.resources.WebAppResourceManagerFactory
Properties	oxf.resources.webapp.rootdir

This resource manager is useful when you want to package an application into a single WAR file for distribution and deployment. The configuration property indicates the path prefix of the resources directory inside a WAR file. It is recommended to store resources under the WEB-INF directory to make sure that the resources are not exposed to remote clients.

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-value>
    org.orbeon.oxf.resources.WebAppResourceManagerFactory
  </param-value>
</context-param>
<context-param>
  <param-name>oxf.resources.webapp.rootdir</param-name>
  <param-value>/WEB-INF/resources</param-value>
</context-param>
```

3.1.6. URL Resource Manager

Purpose	Loading resources from a URL
Factory	org.orbeon.oxf.resources.URLResourceManagerFactory
Properties	oxf.resources.url.base

This resource manager is able to load resources from any URL (http or ftp). It can be used if your resources are located on a web server or a content management system with an HTTP interface.

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-value>org.orbeon.oxf.resources.URLResourceManagerFactory</param-value>
</context-param>
<context-param>
  <param-name>oxf.resources.url.base</param-name>
  <param-value>http://www.somewhere.com/base/</param-value>
</context-param>
```

3.1.7. WebDAV Resource Manager

Purpose	Loading resources from a WebDAV repository
Factory	org.orbeon.oxf.resources.WebDAVResourceManagerFactory
Properties	oxf.resources.webdav.base oxf.resources.webdav.username oxf.resources.webdav.password

This resource manager is able to load resources from a WebDAV repository through HTTP. Examples of WebDAV repositories include:

- Apache Jakarta Slide
- Subversion

For more information about WebDAV, please consult [WebDAV Resources](#).

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-value>
```



```

    org.orbeon.oxf.resources.WebDAVResourceManagerFactory
  </param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.webdav.base</param-name>
  <param-val ue>http://www.somewhere.com/base</param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.webdav.username</param-name>
  <param-val ue>joe</param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.webdav.password</param-name>
  <param-val ue>password</param-val ue>
</context-param>

```

Warning

The WebDAV resource manager is considered experimental.

3.1.8. Priority Resource Manager

Purpose	Chains several resource managers in order
Factory	org.orbeon.oxf.resources.PriorityResourceManagerFactory
Properties	oxf.resources.priority.1 oxf.resources.priority.2 oxf.resources.priority.n

With the priority resource manager you can chain several resource managers. It is crucial to be able to load resources from multiple sources since some resources are bundled in the OPS JAR file. Thus, the class loader resource manager must always be in the priority chain. It usually has the lowest priority so the application developer can override system resources.

There can be any number of chained resource managers. They are configured by adding a `oxf.resources.priority.n` property, where `n` is an integer.

Warning

The priority resource manager is more efficient when most resources are found in the first resource manager specified.

```

<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-val ue>
    org.orbeon.oxf.resources.PriorityResourceManagerFactory
  </param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.webapp.rootdir</param-name>
  <param-val ue>/WEB-INF/resources</param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.priority.1</param-name>
  <param-val ue>
    org.orbeon.oxf.resources.WebAppResourceManagerFactory
  </param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.priority.2</param-name>
  <param-val ue>
    org.orbeon.oxf.resources.ClassLoaderResourceManagerFactory
  </param-val ue>
</context-param>

```

3.1.9. Caché Database Resource Manager

Purpose	Loading resources from the Caché database
Factory	org.orbeon.oxf.resources.CacheResourceManagerFactory
Properties	oxf.resources.cache.url oxf.resources.cache.username oxf.resources.cache.password

This resource manager loads XML documents from [InterSystems Caché Object Database](#). `eXtc` provides a W3C DOM interface to Caché and allows OPS to store and retrieve XML documents. This section assumes Caché and `eXtc` are already installed on your system.

Note

To enable this Resource Manager, you must copy `CacheDB.jar` from Caché distribution into OPS `WEB-INF/lib` directory.

OPS ships with `CacheImport` to populate the database with resource from a directory. Set your `CLASSPATH` environment variable to include the following JAR files to run `CacheImport`:

- ops.jar
- CacheDB.jar
- commons-cli.jar
- xercesImpl-2_2_1_orbeon.jar
- xml-apis-2_5_1.jar
- dom4j-1_4.jar

The options below are available:

Option	Meaning
-u,--url arg	Caché URL
-l,--login arg	Caché Login
-p,--password arg	Caché Password
-r,--root arg	Resource Root

You run `CacheImport` with the following command:

```
java org.orbeon.oxf.resources.CacheImport
```

```
<context-param>
  <param-name>oxf.resources.factory</param-name>
  <param-val ue>
    org.orbeon.oxf.resources.CacheResourceManagerFactory
  </param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.cache.url</param-name>
  <param-val ue>jdbc:Cache://localhost:1972/OXF</param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.cache.username</param-name>
  <param-val ue>_SYSTEM</param-val ue>
</context-param>
<context-param>
  <param-name>oxf.resources.cache.password</param-name>
  <param-val ue>sys</param-val ue>
</context-param>
```

3.2. Properties File

3.2.1. Overview

- **Rationale** – The OPS properties are used by some processors to configure or customize their behavior. This section describes how the system is configured, and lists all the customizable properties.
- **Properties File Location** – The properties sub-system is initialized after the [Resource Manager](#) (the properties being read like any other OPS resources). By default it tries to load a file from the URL `oxf:/properties.xml`. This value can be overridden in the Web application descriptor `web.xml` with the `oxf.properties` initialization parameter:

```
<context-param>
  <param-name>oxf.properties</param-name>
  <param-val ue>oxf:/config/properties.xml</param-val ue>
</context-param>
```

- **Automatic Reloading** – The property file is reloaded every time it is changed, however some properties are taken into account only when the server is first started.
- **Property types** – Properties have a type, which must be one of the following XML Schema simple types: `xs:anyURI`, `xs:integer`, `xs:boolean`, `xs:QName`, `xs:string`, `xs:date`, `xs:dateTime`.
- **Global and Processor Properties** – There are two types of properties: global properties that apply to the system as a whole, and processor-specific properties. For instance, you set the cache size with a global property:

```
<property as="xs:integer" name="oxf.cache.size" val ue="200"/>
```

On the other hand setting the maximum amount of bytes that can be uploaded to the server is set with a processor specific property. Note the additional `processor-name` attribute:

```
<property as="xs:integer" processor-name="oxf:request" name="max-upload-size" val ue="10000000"/>
```

3.2.2. Global Properties

Purpose	Configures the logging system
Type	xs:anyURI
Default Value	The logging system not initialized with a warning if this property is not present.

OPS uses the Log4J logging framework. Log4J is configured with an XML file conforming to the [Log4J DTD](#). Here is a sample Log4J configuration:

```
<log4j:configuration xmlns:log4j="log4j">
  <appender name="ConsoleAppender" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x - %m%n"/>
    </layout>
  </appender>
  <category name="org.orbeon.oxf.processor.DebugProcessor">
    <priority value="info"/>
  </category>
</root>
<root>
  <priority value="error"/>
  <appender-ref ref="ConsoleAppender"/>
</root>
</log4j:configuration>
```

If this property is not set, the Log4J initialization is skipped. This is useful if another subsystem of your application has already initialized Log4J prior to the loading of OPS.

Purpose	Setup the size of the OPS cache
Type	xs:integer
Default Value	200

OPS uses an efficient caching system. OPS automatically determines what can be cached and when to expire objects. The cache has a default size of 200, meaning that it can hold 200 objects. This size is reasonable for most applications. A bigger cache tends to make the application faster, but it uses more memory. To tune the cache, start the Java VM with the `-verbose:gc` option. Then with the application under typical load, look at the server logs and monitor:

- How often full garbage collections happen
- The value of the "success rate" reported by OPS after each page is generated

The "success rate" is the percentage of the objects needed to generate the pages that were found in the cache. The closer this value is to 100%, the better. If the success rate is too low (say below 70% with a mostly static site), then the cache size should be increased. If full garbage collections happen too often, then the VM heap size should be increased. On the other hand, if the success rate is acceptable and garbage collections do not happen too often, you can consider reducing the size of the cache and maybe even the heap size.

Tuning the VM heap size and cache size is not always needed and in most cases the default values are acceptable.

Purpose	Controls the automatic processor validation
Type	xs:boolean
Default Value	Enabled

Many processors validate their configuration input with a schema. This validation is automatic and allows meaningful error reporting. To potentially improve the performance of the application, validation can be disabled in production environments. It is however strongly discouraged to do so, as validation can highly contribute to the robustness of the application.

Purpose	Controls user-defined validation
Type	boolean
Default Value	Enabled

User-defined validation is activated in the [XML Pipeline Definition Language](#) with the attributes `schema-href` and `schema-uri`. To potentially improve the performance of the application, validation can be disabled in production environments. It is however strongly discouraged to do so, as validation can highly contribute to the robustness of the application.

Purpose	Defines OPS processors
Type	xs:anyURI
Default Value	A default prologue is loaded automatically

Loads a OPS prologue file, where processors are declared. OPS is bundled with a default prologue file containing all processors. You can create processors (see the [processor API](#)) and declare them in a custom prologue. The custom prologue doesn't replace, but completes the default prologue. It is possible to override a default processor with a custom implementation by binding it to the same URI as the default processor. The following example shows a simple custom prologue declaring an hypothetical processor:

```
<processors>
  <processor name="oxf:myprocessor">
    <class name="com.company.oxf.MyProcessor"/>
  </processor>
</processors>
```

Purpose	Enable inspection SAX events
Type	xs:boolean
Default Value	false

SAX is the underlying mechanism in OPS by which processors receive and generate XML data. Given only the constraints of the SAX API, it is possible for a processor to generate an invalid sequence of SAX events. Another processor that receives that invalid sequence of events may or may not be able to deal with it without throwing an exception. Some processors try to process invalid SAX events, while others throw exceptions. This means that when a processor generating an invalid sequence of SAX events is used in a pipeline, the problem might go unnoticed, or it might cause some other processor downstream to throw an exception.

To deal more efficiently with those cases, the `sax.inspection` property can be set to `true`. When it is set to `true`, the pipeline engine checks the outputs of every processor at runtime and makes sure that valid SAX events are generated. When an error is detected, an exception is thrown right away, with information about the processor that generated the invalid SAX events.

There is a performance penalty for enabling SAX events inspection. So this property should not be enabled on a production system.

Purpose	Specify the name of a class that implements the interface <code>org.orbeon.oxf.pipeline.api.PipelineContext.Trace</code> .
Type	xs:NCName
Default Value	None

Note

OPS ships with two implementations of `Trace`, `org.orbeon.oxf.processor.NetworkTrace` and `org.orbeon.oxf.processor.StdoutTrace`. `NetworkTrace` sends profiling information to Studio which displays the data in the trace views. `StdOutTrace` simply dumps profiling information to standard out.

Purpose	Specify the host name that <code>org.orbeon.oxf.processor.NetworkTrace</code> will send data to.
Type	xs:NMTOKEN
Default Value	localhost

Purpose	Specify the port that <code>org.orbeon.oxf.processor.NetworkTrace</code> will send data to.
Type	xs:nonNegativeInteger
Default Value	9191

3.2.3. Java Processor Properties

Name	classpath
Purpose	Defines a directory where Java class files are located. The Java processor dynamically compiles Java code, and may need some libraries. This property defines the classpath used by the compiler.
Processor name	oxf:java
Type	xs:string
Default Value	None

Name	jarpath
Purpose	Defines a list of directories where JAR files are located. The Java processor dynamically compiles Java code, and may need some libraries. This property defines a "JAR path", a list of directories containing JAR files that will be added to the classpath when compiling and running the processor executed by the Java processor.

Processor name	oxf:java
Type	xs:string
Default Value	None

Name	compiler-jar
Purpose	Define a URL pointing to a JAR file containing the Java compiler to use. If this property is set, the Java processor adds the specified JAR file to the class path used to search for the main compiler class.
Processor name	oxf:java
Type	xs:anyURI
Default Value	If the property is not specified, the Java processor tries to load the main compiler class first using the current class loader. If this fails, it retrieves the <code>java.home</code> system property which specifies a directory on disk. If that directory is called <code>jre</code> , and there exists a JAR file relative to that directory under <code>../lib/tools.jar</code> , that JAR file is added to the class path used to search for the main compiler class. This covers most cases where the standard Sun JDK is used, so that the <code>compiler-jar</code> property does not have to be specified.

Name	compiler-class
Purpose	Define a class name containing the Java compiler to use. The Java processor loads the corresponding class and calls a static method on this class with the following signature: <code>public static int compile(String[] commandLine, PrintWriter printWriter)</code> .
Processor name	oxf:java
Type	xs:string
Default Value	com.sun.tools.javac.Main

3.2.4. Email Processor Properties

The following property can be specified globally as a property instead of being part of the processor configuration:

Name	smtp-host
Purpose	Configure the SMTP host for all email processors. This global property can be overridden by local processor configurations.
Processor name	oxf:email
Type	xs:string
Default Value	None

The following properties can be used for testing purposes:

Name	test-smtp-host
Purpose	Configure a test SMTP host for all email processors. This global property when specified overrides all the other SMTP host configurations for all Email processors, whether in the processor configuration or using the <code>smtp-host</code> property.
Processor name	oxf:email
Type	xs:string
Default Value	None

Note

This property replaces the deprecated `smtp` property.

Name	test-to
Purpose	Configure a test recipient email address for all email processors. This global property when specified overrides all the other SMTP recipient configurations for all Email processors.

Processor name	oxf:email
Type	xs:string
Default Value	None

Note

This property replaces the deprecated `forceto` property.

For example, those properties can be used as follows:

```
<property as="xs:string" processor-name="oxf:email" name="smtp-host" value="mail.example.org"/>
<property as="xs:string" processor-name="oxf:email" name="test-smtp-host" value="test.example.org"/>
<property as="xs:string" processor-name="oxf:email" name="test-to" value="joe@example.org"/>
```

The test properties can easily be commented out for deployment.

3.3. Non-XML Documents in XPL

3.3.1. Introduction

In OPS [XPL](#) and [pipelines](#) only deal with XML documents. This means that between processor outputs and processor inputs in a pipeline, only pure XML infosets circulate. There is however often a need to handle non-XML data in pipelines, in particular:

- **Binary document:** any document that can be represented as a stream of bytes. In general this is the case of any document, but some document formats are almost always represented this way: images, sounds, PDF documents, etc.
- **Text documents:** any document that can be represented as a stream of characters. Some documents are better looked at this way, like plain txt files, HTML files, and even the textual representation of XML.

OPSs addresses this question by defining two standard XML document formats to embed binary and text documents within an XML infoset. This solution has the benefit of keeping XPL simple by limiting it to pure XML infosets, while allowing XPL to conveniently manipulate any binary and text document.

3.3.2. Binary Documents

A binary document consist of a `document` root node containing character data encoded with Base64. An `xsi:type` attribute is also present, as well as an optional `content-type` attribute, for example:

```
<document xsi:type="xs:base64Binary" content-type="image/jpeg">
  /9j /4AAQSkZJRgABAQEBygHKAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwsHBwYGBw4KCggLEA4R ...
  KKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooA /2Q==
</document>
```

Note

For the curious, the Base64 encoding is documented in [RFC 2045](#). This encoding represents binary data by mapping it to a set of 64 ASCII characters.

Such documents are not meant to be read by users, in the same way that regular binary files are not meant to be examined by users. Binary documents are generated by OPS processors, like the [URL generator](#) and [converters](#). They are consumed by processors like the [HTTP serializer](#), the [Email processor](#), and [converters](#).

3.3.3. Text Documents

A text document consists of a `document` root element containing the text. An `xsi:type` attribute is also present, as well as an optional `content-type` attribute:

```
<document xsi:type="xs:string" content-type="text/plain">
  This is line one of the input document! This is line two of the input
  document!
</document>
```

The `content-type` attribute may have a `charset` parameter providing a hint for the character encoding, for example:

```
<document xsi:type="xs:string" content-type="text/plain; charset=iso-8859-1">
  This is line one of the input document! This is line two of the input document! This is line three of the input
  document!
</document>
```

Because XML character data itself is represented in Unicode (in other words it is designed to allow representing in a same document all the characters specified by the Unicode specification), there is no requirement for specifying character encoding in XML pipelines. However, when an XML infoset is read or written as a textual XML document, specifying a character encoding may be a useful hint. For example a URL generator can, with this mechanism, communicate to an HTTP serializer the preferred character encoding obtained when the document was read. The serializer may then use that hint, but it is by no means authoritative.

In general, XML documents can be read and written using the `utf-8` character encoding, which allows representing all the Unicode characters. However, when dealing with other types of text documents, tools such as text editors may not be able to deal correctly with `utf-8`. In such cases, it can be useful to use even more widespread character encodings such as `iso-8859-1` or `us-ascii`. The drawback is that such encodings allow representing a much smaller set of characters than `utf-8`.

Unlike binary documents, text documents can easily be examined by users. They can also be easily manipulated by languages such as [XSLT](#). Like binary documents, they are generated by OPS processors, like the [URL generator](#) and [converters](#). They are consumed by processors like the [HTTP serializer](#), the [Email processor](#), and [converters](#).

3.3.4. Streaming

Processors can stream binary and text documents by issuing a number of short character SAX events. It is therefore possible to generate "infinitely" long binary and text documents with a constant amount of memory, assuming both the sender and the receiver of the document are able to perform streaming. This is the case for example of the [URL generator](#) and the [HTTP serializer](#).

3.4. Error Processor

3.4.1. Rationale

When OPS encounters an error, it throws two kinds of exceptions: `OXFException` and its subclass, `ValidationException`. They both contain a nested root cause, and `ValidationException` contains document location information.

When an exception is thrown, OPS displays a default error page containing the root cause and location information if available, as well as a detailed OPS stack trace. However, the application developer can display a different page by specifying a processor to be executed upon error. This processor could be, for example, a pipeline logging the error and displaying a nice error page to the user.

3.4.2. Configuration

The error processor is configured in `web.xml`. For more information, please refer to the [Packaging and Deployment](#) documentation.

3.4.3. Example

The following code in `web.xml` specifies that upon error, the Pipeline processor must run using the `oxf:/config/error.xpl` pipeline:

```
<servlet>
  <servlet-name>oxf</servlet-name>
  <servlet-class>org. orbeon. oxf. servlet. OPSServlet</servlet-class>
  <!-- The error processor that OXFServlet must execute -->
  <init-param>
    <param-name>oxf. error-processor. name</param-name>
    <param-value>{http://www. orbeon. com/oxf/processors}pipeline</param-value>
  </init-param>
  <!-- The pipeline to execute -->
  <init-param>
    <param-name>oxf. error-processor. input. config</param-name>
    <param-value>oxf: /config/error. xpl </param-value>
  </init-param>
</servlet>
```

In most cases, the custom error processor logs or displays the exception that occurred. A simple error pipeline is shown below, using the [Exception generator](#) to display the root cause message, but not the exception stack trace.

```
<p: config xmlns:p="http://www. orbeon. com/oxf/pipeline">
  <!-- Generate exception document -->
  <p: processor name="oxf: exception">
    <p: output name="data" id="exception"/>
  </p: processor>
  <!-- Apply stylesheet -->
  <p: processor name="oxf: xslt">
    <p: input name="data" href="#exception"/>
    <p: input name="config">
      <xsl: stylesheet version="1.0" xmlns:xsl="http://www. w3. org/1999/XSL/Transform">
        <xsl: template match="/">
          <html>
            <head>
              <title>OPS - Custom Error Page</title>
            </head>
            <body>
              <h1>An error occurred:</h1>
              <p>
                The screen demonstrates a custom error pipeline. For this example, only the message of
                the first exception is displayed.
              </p>
            </body>
          </html>
        </xsl: template>
      </xsl: stylesheet>
    </p: input>
  </p: processor>
</p: config>
```

```

        <p>
            <code>
                <xsl: value-of select="/exceptions/exception[1]/message"/>
            </code>
        </p>
    </body>
</html>
</xsl: template>
</xsl: stylesheet>
</p: input>
<p: output name="data" id="document"/>
</p: processor>
<!-- Get some request information -->
<p: processor name="oxf: request">
    <p: input name="config">
        <config>
            <include>/request/container-type</include>
            <include>/request/request-path</include>
        </config>
    </p: input>
    <p: output name="data" id="request"/>
</p: processor>
<!-- Apply theme -->
<p: processor name="oxf: xslt">
    <p: input name="data" href="#document"/>
    <p: input name="request" href="#request"/>
    <p: input name="config" href="oxf:/config/theme/theme.xsl"/>
    <p: output name="data" id="themed"/>
</p: processor>
<!-- Rewrite all URLs in XHTML documents -->
<p: processor name="oxf: xhtml-rewrite">
    <p: input name="rewrite-in" href="#themed"/>
    <p: output name="rewrite-out" id="rewritten-data"/>
</p: processor>
<!-- Convert to HTML -->
<p: processor name="oxf: qname-converter">
    <p: input name="config">
        <config>
            <match>
                <uri>http://www.w3.org/1999/xhtml</uri>
            </match>
            <replace>
                <uri/>
                <prefix/>
            </replace>
        </config>
    </p: input>
    <p: input name="data" href="#rewritten-data"/>
    <p: output name="data" id="html-data"/>
</p: processor>
<p: processor name="oxf: html-converter">
    <p: input name="config">
        <config>
            <public-doctype>-//W3C//DTD HTML 4.01 Transitional//EN</public-doctype>
            <version>4.01</version>
            <encoding>utf-8</encoding>
        </config>
    </p: input>
    <p: input name="data" href="#html-data"/>
    <p: output name="data" id="converted"/>
</p: processor>
<!-- Serialize -->
<p: processor name="oxf: http-serializer">
    <p: input name="config">
        <config>
            <status-code>500</status-code>
            <header>
                <name>Cache-Control</name>
                <value>post-check=0, pre-check=0</value>
            </header>
        </config>
    </p: input>
    <p: input name="data" href="#converted"/>
</p: processor>
</p: config>

```

3.5. Listeners

3.5.1. Servlet Context Listener

The OPS Servlet Context Listener allows configuring one processor to be called when the Servlet context is initialized, and one to be called when the Servlet context is destroyed. These processors are looked up in the following locations, in this order:

1. properties.xml;


```
<properties>
  <!-- Other properties -->
  ...
  <!-- Servlet Context Listener properties -->
  <property as="xs:QName" name="oxf.context-initialized-processor.name" value="oxf:pipeline"/>
  <property as="xs:anyURI" name="oxf.context-initialized-processor.input.config" value="oxf:/context/context-
  initialized.xpl"/>
  <property as="xs:QName" name="oxf.context-destroyed-processor.name" value="oxf:pipeline"/>
  <property as="xs:anyURI" name="oxf.context-destroyed-processor.input.config" value="oxf:/context/context-
  destroyed.xpl"/>
</properties>
```

2. Context parameters in web.xml:

```
<context-param>
  <param-name>oxf.context-initialized-processor.name</param-name>
  <param-value>{http://www.orbeon.com/oxf/processors}pipeline</param-value>
</context-param>
<context-param>
  <param-name>oxf.context-initialized-processor.input.config</param-name>
  <param-value>oxf:/context/context-initialized.xpl</param-value>
</context-param>
<context-param>
  <param-name>oxf.context-destroyed-processor.name</param-name>
  <param-value>{http://www.orbeon.com/oxf/processors}pipeline</param-value>
</context-param>
<context-param>
  <param-name>oxf.context-destroyed-processor.input.config</param-name>
  <param-value>oxf:/context/context-destroyed.xpl</param-value>
</context-param>
```

Not every processor can be run from those pipelines, because the execution context is limited. In particular, you can't call processors like the Request generator or the HTTP serializer. You do however have access to the Application context.

For the OPS Servlet Context Listener to be called, you need to configure the following listener in your web.xml:

```
<listener>
  <listener-class>org.orbeon.oxf.webapp.OPSServletContextListener</listener-class>
</listener>
```

Note

There are no default processors. If no processor is specified, no processor is run on context initialization or destruction. This allows configuring a listener with only an initialization pipeline, only a destruction pipeline, both, or none.

The Servlet Context Listener logs its actions at level `info`. This can be controlled in `log4j.xml`:

```
<category name="org.orbeon.oxf.webapp.OPSServletContextListener">
  <priority value="info"/>
</category>
```

3.5.2. Session Listener

The OPS Session Listener allows configuring one processor to be called when the Session is created, and one to be called when the Session is destroyed. These processors are looked up in the following locations, in this order:

1. properties.xml:

```
<properties>
  <!-- Other properties -->
  ...
  <!-- Session Listener properties -->
  <property as="xs:QName" name="oxf.session-created-processor.name" value="oxf:pipeline"/>
  <property as="xs:anyURI" name="oxf.session-created-processor.input.config" value="oxf:/context/session-
  created.xpl"/>
  <property as="xs:QName" name="oxf.session-destroyed-processor.name" value="oxf:pipeline"/>
  <property as="xs:anyURI" name="oxf.session-destroyed-processor.input.config" value="oxf:/context/session-
  destroyed.xpl"/>
</properties>
```

2. Context parameters in web.xml:

```
<context-param>
  <param-name>oxf.session-created-processor.name</param-name>
  <param-value>oxf:pipeline</param-value>
</context-param>
```

```

<context-param>
  <param-name>oxf.session-created-processor.input.config</param-name>
  <param-value>oxf:/context/session-created.xpl</param-value>
</context-param>
<context-param>
  <param-name>oxf.session-destroyed-processor.name</param-name>
  <param-value>oxf:pipeline</param-value>
</context-param>
<context-param>
  <param-name>oxf.session-destroyed-processor.input.config</param-name>
  <param-value>oxf:/context/session-destroyed.xpl</param-value>
</context-param>

```

Not every processor can be run from those pipelines, because the execution context is limited. In particular, you can't call processors like the Request generator, or any HTTP serializers. You have access to the Application and Session contexts.

For the OPS Session Listener to be called, you need to configure the following listener in your `web.xml`:

```

<listener>
  <listener-class>org.orbeon.oxf.webapp.OPSSessionListener</listener-class>
</listener>

```

Note

There are no default processors. If no processor is specified, no processor is run on session creation or destruction. This allows configuring a listener with only a creation pipeline, only a destruction pipeline, both, or none.

The Session Listener logs its actions at level `info`. This can be controlled in `log4j.xml`:

```

<category name="org.orbeon.oxf.webapp.OPSSessionListener">
  <priority value="info"/>
</category>

```

3.6. URL Rewriting

3.6.1. Rationale

Traditionally, Web applications developers generate URLs in their Web pages manually. For example, an HTML `<a>` element contains an `href` attribute specifying the destination of a link. The default OPS Page Flow Controller epilogue contains a URL rewriting mechanism for HTML and XHTML documents.

Absolute URLs (starting with a scheme such as `http:`) are usually reserved to refer to external sites or applications. When referring to the current application, relative URLs, in the form of relative paths or absolute paths, are commonly used. Developers must make sure that the URL interpreted by the Web browser and the application server refers to the correct page or resource:

- **Relative paths:** such paths are interpreted by the Web browser as relative to a URL base, usually the URL of the page being requested, unless specified differently. For example, if a browser requests `/oxf/example1/page1`:

Relative Path	Resulting Absolute Path
<code>page2</code>	<code>/oxf/example1/page2</code>
<code>../page3</code>	<code>/oxf/page3</code>
<code>../example2/page4</code>	<code>/oxf/example2/page4</code>

Using relative paths raises the issue that if a page is moved, all the links within that page have to be changed.

- **Absolute paths:** such paths start with a `"/`. The developer can generate the same paths as above by directly writing the resulting absolute paths. The issue with this solution is that it is necessary to write the exact absolute path, often including a J2EE Web applications context path such as `/oxf`. Hardcoding the context path in every URL makes it impossible to change the application context without changing all the URLs in the application. To alleviate this issue, developers often use relative URLs, with the problem mentioned above.

The issue is even more important with Java Portlets (JSR-168), as URLs must be generated by calling a specific Java API. With page template languages such as JSP, this is done using tag libraries. All the pages in an application must be modified when moved from a deployment as a Servlet to a deployment as a Portlet.

A solution to the issues mentioned above is to post-process all URLs to make the developer's life easier.

3.6.2. Default URL Rewriting

This section describes the default URL rewriting implementation in Presentation Server. It is implemented in the processors `oxf:xhtml-rewrite` and `oxf:html-rewrite`.

The `form`, `a`, `link`, `img`, and `input` elements are rewritten. In addition, their XHTML counterparts in the `http://www.w3.org/1999/xhtml` namespace are also rewritten. URLs can be parsed by the rewriting algorithm, so developers have to make sure that they are well-formed. Absolute URLs (with a scheme) are left unmodified. The special case of URLs starting with a query string (e.g. `?name=value`) is handled. This last syntax is supported by most Web browsers and because of its convenience, it is supported by the default rewriting algorithm as well.

Element / Attribute	Action
<code>form/@action</code>	If the URL is a relative path, it is left unchanged. If the URL is an absolute path, the context path is pre-pended. Absolute URLs are left unchanged.
<code>a/@href</code>	
<code>link/@href</code>	
<code>img/@src</code>	
<code>input[@type='image']/@src</code>	
<code>script/@src</code>	

Element / Attribute	Action
<code>form/@action</code>	Rewritten to an action URL using the Portlet API method <code>RenderResponse.createActionURL()</code> . The resulting URL results in an action URL targeting the current portlet. Absolute URLs are left unchanged.
<code>form/@method</code>	If no <code>form/@method</code> is supplied, an HTTP <code>POST</code> is forced, because the Portlet specification recommends submitting forms with <code>POST</code> . If a method is supplied, the method is left unchanged.
<code>a/@href</code>	Rewritten to a render URL using the Portlet API method <code>RenderResponse.createRenderURL()</code> . The resulting URL results in a render URL targeting the current portlet. Absolute URLs are left unchanged.
<code>img/@src</code>	Rewritten to a resource URL encoding. The resulting URL points to a resource within your Web application. Absolute URLs are left unchanged.
<code>input[@type='image']/@src</code>	
<code>script/@src</code>	
<code>link/@href</code>	
<code>script</code> and <code>SCRIPT</code>	In text within those elements or their XHTML counterparts in the <code>http://www.w3.org/1999/xhtml</code> namespace, occurrences of the string <code>wsrp_rewritewsrp_rewrite_</code> are replaced with the Portlet namespace as obtained by the Portlet API method <code>RenderResponse.encodeNamespace(null)</code> .

3.6.3. Working with URL Rewriting

In OPS, URLs come from two different sources:

- Within XSLT templates, for example single pages or a theme stylesheet. This is the case for most URLs, including links to resources (images, CSS stylesheets, JavaScript files, etc.), links to other pages, etc.
- Within an XForms' `submission-info` element, to specify the action to be called when a form is submitted.

In both cases, URLs are subject to the following sections.

URLs can be written as relative paths as usual, but they can also be written as absolute paths without concerns about the context path. For example:

Initial Path	Resulting Path
<code>/example1/page2</code>	<code>/oxf/example1/page2</code>
<code>/page3</code>	<code>/oxf/page3</code>
<code>/example2/page4</code>	<code>/oxf/example2/page4</code>

With Portlets, the benefit is even greater. Write your URLs as you would in a regular Servlet-based application, and the rewriting pipeline takes care of calling the Portlet API to encode the URLs. Since portlets do not have the concept of path, URL paths are encoded as a special parameter named `oxf.path`. Relative paths are resolved against the current path. The following table illustrates action URL and render URL rewriting:

Initial Path	Resulting Portlet Parameters
--------------	------------------------------

/example1/page1?name1=value1&name2=value2	<ul style="list-style-type: none"> ■ <code>oxf.path=/example1/page1</code> ■ <code>name1=value1</code> ■ <code>name2=value2</code>
?name1=value1&name2=value2	<ul style="list-style-type: none"> ■ <code>name1=value1</code> ■ <code>name2=value2</code>
Assuming the current value of <code>oxf.path</code> is <code>/example1/page1</code> : ../example2/page2?name1=value1&name2=value2	<ul style="list-style-type: none"> ■ <code>oxf.path=/example2/page2</code> ■ <code>name1=value1</code> ■ <code>name2=value2</code>

The following table illustrates resource URL rewriting:

Initial Path	Resulting Path
/path/to/my/image.gif?scale=100	/oxf/path/to/my/image.gif?scale=100
Assuming the current value of <code>oxf.path</code> is <code>/example1/page1</code> : my/image.gif?scale=100	/oxf/example1/my/image.gif?scale=100 Note that using resource URLs relative to an OPS Portlet path (as handled by OPS) does not necessarily make sense unless you define your hierarchy of resources carefully.

3.6.4. Known Limitations

- The input document should not contain:
 - Elements and attribute containing the string `wsrp_rewritewsrp_rewrite`
 - Namespace URIs containing the string `wsrp_rewritewsrp_rewrite`
 - Processing instructions containing the string `wsrp_rewritewsrp_rewrite`
- It is not possible to specify:
 - A destination portlet mode
 - A destination window state
 - A secure URL

3.7. XML Namespaces

3.7.1. Introduction

Namespaces play an important role in XML applications. In particular, they allow for modularity and for mixing different XML vocabularies in a single document, for example XHTML and XForms.

For more information, please visit the [XML Namespaces specification](#).

3.7.2. Namespace Usage in OPS

The number of XML namespaces used in OPS is quite large, and it is easy for developers to lose track of which is which. The table below summarizes the usage of XML namespaces in OPS, with links to the relevant documentation and specifications.

Namespace URI	Usual Prefix	Usage	Example	Specification
http://www.orbeon.com/oxf/pipeline	p	XML Pipeline Language program (XPL)	<code>p:processor</code>	XML Pipeline Language
http://www.orbeon.com/oxf/processors	oxf	Standard OPS processors referred from XPL programs	<code>oxf:http-serializer</code>	OPS processors documentation
http://orbeon.org/oxf/xml/xforms	xxforms	OPS extensions to XForms	<code>xxforms:appearance</code>	OPS XForms Reference
http://orbeon.org/oxf/xml/formatting	f	OPS view formatting	<code>f:xml-source</code>	
http://orbeon.org/oxf/xml/portlet	portlet	OPS portlet tagging	<code>portlet:is-portlet-content</code>	

http://www.orbeon.com/xslt-function	function	OPS XSLT functions	function:evaluate	
http://orbeon.org/oxf/xml/local	local	User-defined XSLT functions	local:my-function	
http://orbeon.org/oxf/xml/xmldb	xdb	XML:DB processors configurations	xdb:query	XML:DB Processors
http://orbeon.org/oxf/xml/sql	sql	SQL procesor configuration	sql:get-column	SQL Processor
http://orbeon.org/oxf/xml/datatypes	odt	SQL procesor XML data types.	odt:xmlFragment	SQL Processor
http://www.w3.org/2001/XInclude	xi	XInclude elements <div> Note The XInclude processor uses this (correct) namespace instead of http://www.w3.org/2003/XInclude, which is used by some XML parsers (see next entry). </div>	xi:include	XML Inclusions (XInclude) Version 1.0
http://www.w3.org/2003/XInclude	xi	XInclude elements <div> Note The official namespace to use for XInclude 1.0 is http://www.w3.org/2001/XInclude (that is, with a 2001 in it). However, currently in OPS this URI will generate warnings from the Xerces XML parser which incorrectly prefers using a URI with 2003. This only occurs when XInclude is processed at parsing time (see the URL generator but not when the XInclude processor is used . </div>	xi:include	XML Inclusions (XInclude) Version 1.0
http://www.w3.org/1999/XSL/Transform	xsl	XSLT 1.0 or 2.0 stylesheet.	xsl:transform	XSL Transformations (XSLT) Version 1.0 XSL Transformations (XSLT) Version 2.0
http://www.w3.org/2004/07/xpath-datatypes	xdt	XPath 2.0 datatypes <div> Note This URI changes with each release of the XPath 2.0 draft specification, and with each release of the Saxon XSLT processor. </div>	xdt:dayTimeDuration	XML Path Language (XPath) 2.0
http://saxon.sf.net/	saxon	Saxon XSLT processor extensions	saxon:parse	Saxon Extensions
http://www.w3.org/2001/XMLSchema	xs	XML Schema	xs:schema	XML Schema Part 0: Primer Second Edition XML Schema Part 1: Structures Second Edition XML Schema Part 2: Datatypes Second Edition
http://www.w3.org/2001/XMLSchema-instance	xsi	XML Schema attributes	xsi:type	XML Schema Part 1: Structures Second Edition
http://www.w3.org/2001/XMLSchema-datatypes	N/A	XML Schema datatypes, used by XML Schema and by Relax NG		XML Schema Part 2: Datatypes Second Edition
http://www.xmldb.org/xupdate	xu	XUpdate program	xu:modifications	OPS Page Flow

http://www.w3.org/2002/xforms	xforms	XForms markup	xforms:input	XForms 1.0 OPS XForms Reference
http://www.w3.org/2001/xml-events	ev	XML Events (used by XForms)	ev:event	OPS XForms Reference
http://www.w3.org/1999/xhtml	xhtml	XHTML markup	xhtml:body	XHTML 1.0
http://relaxng.org/ns/structure/1.0	N/A	Relax NG 1.0 schema		RELAX NG home page RELAX NG Specification

4. Processors Reference

4.1. URL Generator

4.1.1. Introduction

Generators are a special category of processors that have no XML data inputs, only outputs. They are generally used at the top of an XML pipeline to generate XML data from a Java object or other non-XML source.

The URL generator fetches a document from a URL and produces an XML output document. Common protocols such as `http:`, `ftp:`, and `file:` are supported as well as the OPS resource protocol (`oxf:`). See [Resource Managers](#) for more information about the `oxf:` protocol.

4.1.2. Content Type

The URL generator operates in several modes depending on the content type of the source document. The content type is determined according to the following priorities:

1. Use the content type in the `content-type` element of the configuration if `force-content-type` is set to `true`.
2. Use the content type set by the connection (for example, the content type sent with the document by an HTTP server), if any. Note that when using the `oxf:` or `file:` protocol, the connection content type is never available. When using the `http:` protocol, the connection content type may or may not be available depending on the configuration of the HTTP server.
3. Use the content type in the `content-type` element of the configuration, if specified.
4. Use `application/xml`.

4.1.3. XML Mode

The XML mode is selected when the content type is `text/xml`, `application/xml`, or ends with `+xml` according to the selection algorithm above. The generator fetches the specified URL and parses the XML document. If the `validating` option is set to `true`, a validating parser is used, otherwise a non-validating parser is used. Using a validating parser allows to validate a document with a DTD. In addition, the URL generator is able to handle XInclude inclusions during parsing. By default, it does so. This can be disabled by setting the `handle-xinclude` option to `false`.

Example:

```
<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>oxf:/urlgen/note.xml</url>
      <content-type>application/xml</content-type>
      <validating>true</validating>
      <handle-xinclude>false</handle-xinclude>
    </config>
  </p:input>
  <p:output name="data" id="xml"/>
</p:processor>
```

Note

The URL must point to a well-formed XML document. If it doesn't, an exception will be raised.

4.1.4. HTML Mode

The HTML mode is selected when the content type is `text/html` according to the selection algorithm above. In this mode, the URL generator uses [HTML Tidy](#) to transform HTML into XML. This feature is useful to later extract information from HTML using XPath.

Examples:

```
<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://www.cnn.com</url>
      <content-type>text/html</content-type>
    </config>
  </p:input>
  <p:output name="data" id="html"/>
</p:processor>
```

```
<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>oxf:/html/example.html</url>
      <content-type>text/html</content-type>
    </config>
  </p:input>
  <p:output name="data" id="html"/>
</p:processor>
```

```

        <force-content-type>true</force-content-type>
    </config>
</p:input>
<p:output name="data" id="html" />
</p:processor>

```

Note

HTML Tidy has some tolerance for malformed HTML, but it is encouraged to access well-formed HTML whenever possible.

4.1.5. Text Mode

The text mode is selected when the content type according to the selection algorithm above starts with `text/` and is different from `text/html` or `text/xml`, for example `text/plain`. In this mode, the URL generator reads the input as a text file and produces an XML document containing the text read.

Example:

```

<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>oxf:/list.txt</url>
      <content-type>text/plain</content-type>
    </config>
  </p:input>
  <p:output name="data" id="text" />
</p:processor>

```

Assume the input document contains the following text:

```

This is line one of the input document!
This is line two of the input document!
This is line three of the input document!

```

The resulting document consists of a `document` root element containing the text according to the [text document format](#). An `xsi:type` attribute is also present, as well as a `content-type` attribute:

```

<document xsi:type="xs:string" content-type="text/plain">
  This is line one of the input document! This is line two of the input
  document!
</document>

```

Note

The URL generator performs streaming. It generates a stream of short character SAX events. It is therefore possible to generate an "infinitely" long document with a constant amount of memory, assuming the generator is connected to other processors that do not require storing the entire stream of data in memory, for example the [SQL processor](#) (with an appropriate configuration to stream BLOBs), or the [HTTP serializer](#).

4.1.6. Binary Mode

The binary mode is selected when the content type is neither one of the XML content types nor one of the `text/*` content types. In this mode, the URL generator uses a Base64 encoding to transform binary content into XML according to the [binary document format](#). For example:

```

<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>oxf:/my-image.jpg</url>
      <content-type>image/jpeg</content-type>
    </config>
  </p:input>
  <p:output name="data" id="image-data" />
</p:processor>

```

The resulting document consists of a `document` root node containing character data encoded with Base64. An `xsi:type` attribute is also present, as well as a `content-type` attribute, if found:

```

<document xsi:type="xs:base64Binary" content-type="image/jpeg">
  /9j/4AAQSkZJRgABAQEBygHKAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwsHBwYGBw4KCggLEA4R...
  KKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooA/2Q==
</document>

```

Note

The URL generator performs streaming. It generates a stream of short character SAX events. It is therefore possible to generate an "infinitely" long document with a constant amount of memory, assuming the generator is connected to other processors that do not require storing the entire stream of data in memory, for example the [SQL processor](#) (with an appropriate configuration to stream BLOBs), or the [HTTP serializer](#).

4.1.7. Character Encoding

For text and XML, the character encoding is determined as follows:

1. Use the encoding in the `encoding` element of the configuration if `force-encoding` is set to `true`.
2. Use the encoding set by the connection (for example, the encoding sent with the document by an HTTP server), if any, unless `ignore-connection-encoding` is set to `true` (for XML documents, precedence is given to the connection encoding as per RFC 3023). Note that when using the `oxf:` or `file:` protocol, the connection encoding is never available. When using the `http:` protocol, the connection encoding may or may not be available depending on the configuration of the HTTP server. The encoding is specified along with the content type in the `content-type` header, for example: `content-type: text/html; charset=iso-8859-1`.
3. Use the encoding in the `encoding` element of the configuration, if specified.
4. For XML, the character encoding is determined automatically by the XML parser.
5. For text, including HTML: use the default of `iso-8859`

When reading XML documents, the preferred method of determining the character encoding is to let either the connection or the XML parser auto detect the encoding. In some instances, it may be necessary to override the encoding. For this purpose, the `force-encoding` and `encoding` elements can be used to override this default behavior, for example:

```
<p: processor name="oxf:url-generator" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <config>
      <url>oxf:/urlgen/note.xml</url>
      <content-type>application/xml</content-type>
      <encoding>iso-8859-1</encoding>
      <force-encoding>true</force-encoding>
    </config>
  </p: input>
  <p: output name="data" id="xml" />
</p: processor>
```

This use should be reserved for cases where it is known that a document specifies an incorrect encoding and it is not possible to modify the document.

HTML example:

```
<p: processor name="oxf:url-generator" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <config>
      <url>http://www.cnn.com</url>
      <content-type>text/html</content-type>
      <encoding>iso-8859-1</encoding>
    </config>
  </p: input>
  <p: output name="data" id="html" />
</p: processor>
```

Note that only the following encodings are supported for HTML documents:

- `iso-8859-1`
- `utf-8`

Also note that use of the HTML `<meta>` tag to specify the encoding from within an HTML document is not supported.

4.1.8. HTTP Headers

When retrieving a document from an HTTP server, you can optionally specify the headers sent to the server by adding one or more `header` elements, as illustrated in the example below:

```
<p: processor name="oxf:url-generator" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <config>
      <url>http://www.cnn.com</url>
      <content-type>text/html</content-type>
      <header>
        <name>User-Agent</name>
        <value>Mozilla/5.0</value>
      </header>
      <header>
        <name>Accept-Language</name>
        <value>en-us, fr-fr</value>
      </header>
    </config>
  </p: input>
  <p: output name="data" id="html" />
</p: processor>
```

```

    </header>
  </config>
</p:input>
<p:output name="data" id="html" />
</p:processor>

```

4.1.9. Cache Control

It is possible to configure whether the URL generator caches documents locally in the OPS cache. By default, it does. To disable caching, use the `cache-control/use-local-cache` element, for example:

```

<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://www.cnn.com</url>
      <content-type>text/html</content-type>
      <cache-control>
        <use-local-cache>false</use-local-cache>
      </cache-control>
    </config>
  </p:input>
  <p:output name="data" id="html" />
</p:processor>

```

Using the local cache causes the URL generator to check if the document is in the OPS cache first. If it is, its validity is checked with the protocol handler (looking at the last modified date for files, the `last-modified` header for http, etc.). If the cached document is valid, it is used. Otherwise, it is fetched and put in the cache.

When the local cache is disabled, the document is never revalidated and always fetched.

4.1.10. Relative URLs

URLs passed to the URL generator can be relative. For example, consider the following pipeline fragment declared in a file called `oxf:/my-pipelines/backend/import.xpl`:

```

<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>../././documents/claim.xml</url>
    </config>
  </p:input>
  <p:output name="data" id="file" />
</p:processor>

```

In this case, the URL resolves to: `oxf:/documents/claim.xml`.

4.2. Request Generator

4.2.1. Introduction

Generators are a special category of processors that have no XML data inputs, only outputs. They are generally used at the top of an XML pipeline to generate XML data from a Java object or other non-XML source.

The Request generator streams XML from the current HTTP request. It can serialize request attributes including headers, parameters, query strings, user and server information.

Note

The Request generator can be used as the first component in a web application pipeline, but it is recommended to use the [Page Flow Controller](#) and [XForms](#) whenever possible. There are cases where additional data from the request may be required, however, and where the Request generator must be used.

4.2.2. Configuration

The Request generator takes a mandatory configuration to select which request attribute to display. This configuration consists of a series of `include` and `exclude` elements containing XPath expressions selecting a number of element from the request tree. Those expressions can be as complex as any regular XPath 1.0 expression that returns a single node or a node-set. However, it is recommended to keep those expressions as simple as possible. One known limitation is that it is not possible to test on the `value` element of uploaded files, as well as the content of the request body.

Sample Configuration:

```

<config>
  <include>/request/path-info</include>
  <include>/request/headers</include>
  <include>
    /request/parameters/parameter[starts-with(name, 'document-id')]
  </include>

```

```
<excl ude>
  /request/parameters/parameter[name = 'document-id-dummy']
</excl ude>
</config>
```

The full attribute tree is:

```
<request>
  <container-type>servlet</container-type>
  <content-length>-1</content-length>
  <content-type/>
  <parameters>
    <parameter>
      <name>id</name>
      <value>12</value>
    </parameter>
    <parameter>
      <name>print</name>
      <value>false</value>
    </parameter>
  </parameters>
  <body/>
  <protocol>HTTP/1.1</protocol>
  <remote-addr>127.0.0.1</remote-addr>
  <remote-host>localhost</remote-host>
  <scheme>http</scheme>
  <server-name>localhost</server-name>
  <server-port>8080</server-port>
  <is-secure>false</is-secure>
  <auth-type>BASIC</auth-type>
  <remote-user>jdoe</remote-user>
  <context-path>/ops</context-path>
  <headers>
    <header>
      <name>host</name>
      <value>localhost: 8080</value>
    </header>
    <header>
      <name>user-agent</name>
      <value>Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.1) Gecko/20020826
      </value>
    </header>
    <header>
      <name>accept-language</name>
      <value>en-us, en; q=0.50</value>
    </header>
    <header>
      <name>accept-encoding</name>
      <value>gzip, deflate, compress; q=0.9</value>
    </header>
    <header>
      <name>accept-charset</name>
      <value>ISO-8859-1, utf-8; q=0.66, *; q=0.66</value>
    </header>
    <header>
      <name>keep-alive</name>
      <value>300</value>
    </header>
    <header>
      <name>connection</name>
      <value>keep-alive</value>
    </header>
    <header>
      <name>referer</name>
      <value>http://localhost: 8080/ops/</value>
    </header>
    <header>
      <name>cookie</name>
      <value>JSESSIONID=DA6E64FC1E6DFF0499B5D6F46A32186A</value>
    </header>
  </headers>
  <method>GET</method>
  <path-info>/doc/home-welcome</path-info>
  <request-path>/doc/home-welcome</request-path>
  <path-translated>
    C:\orbeon\projects\OPS\build\ops-war\doc\home-welcome
  </path-translated>
  <query-string>id=12&print=false</query-string>
  <requested-session-id>DA6E64FC1E6DFF0499B5D6F46A32186A</requested-session-id>
  <request-uri>/ops/doc/home-welcome</request-uri>
  <request-url>http://localhost: 8888/ops/doc/home-welcome</request-url>
  <servlet-path/>
</request>
```

Note

OPS adds a request attribute: the `request-path`. This attribute is defined as a concatenation of the `servlet-path` and the `path-info`. This is useful because both original attributes are frequently mixed up and often change depending on the application server or its configuration.

Warning

This generator excludes all attributes by default. To obtain the whole attributes tree (as shown in the example above), you must explicitly include `/request`:

```
<p: processor name="oxf:request" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <include>/request</include>
    </config>
  </p:input>
  <p:output name="data" id="request"/>
</p:processor>
```

4.2.3. Request Body

When the request includes `/request/body`, the Request generator retrieves the body of the request sent to the application server. The content of the body is made available as the following data types:

- If the attribute `stream-type` on the `config` element is set to `xs:anyURI`, an URI is returned as the value of the `/request/body` element.
- If the attribute `stream-type` on the `config` element is set to `xs:base64Binary`, the content of the request encoded as Base64 is returned as the value of the `/request/body` element.
- Otherwise, the content of the `/request/body` is set as either `xs:anyURI` if the request body is large (as set by the `max-upload-memory-size` property, by default larger than 10 KB), or `xs:base64Binary` if the request body is small.

Examples of configuration:

```
<config stream-type="xs:anyURI">
  <include>/request/body</include>
</config>
```

```
<config stream-type="xs:base64Binary">
  <include>/request/body</include>
</config>
```

The resulting data type is always set on the body element, for example:

```
<request>
  <body xsi:type="xs:anyURI">file:/C:/Tomcat/temp/upload_00000005.tmp</body>
</request>
```

Warning

Reading the request body is incompatible with reading HTML forms posted with the `multipart/form-data` encoding, typically used when uploading files. In such a case, you should read either only the request body, or only the request parameters.

4.2.4. Uploaded Files

Uploaded files are stored into `parameter` elements, like any other form parameter. The rules for the data type used are the same as for the request body (see above), the data type depending on the `stream-type` attribute and the size of the uploaded files:

```
<config stream-type="xs:anyURI">
  <include>/request/parameters</include>
</config>
```

The `parameter` element for an uploaded file contains the following elements in addition to the `name` and `value` elements use for other parameters:

- `filename`: stores the file name sent by the user agent
- `content-type`: store the media type sent by the user agent
- `content-length`: stores the actual size in bytes of the uploaded data

A resulting uploaded file may look as follows:

```
<request>
  <parameters>
    <parameter>
```

```

<name>upload-form-element-name</name>
<filename>photo.jpg</filename>
<content-type>image/jpeg</content-type>
<content-length>2345</content-length>
<value xsi:type="xs:anyURI">file:/C:/Tomcat/temp/upload_00000005.tmp</value>
</parameter>
</parameters>
</request>

```

Warning

The URL stored as the value of the upload or request body is temporary and only valid for the duration of the current request. It is only accessible from the server side, and will not be accessible from a client such as a web browser. It is not guaranteed to be a `file:` URL, only that it can be read with OPS's [URL generator](#).

4.3. Other Generators

4.3.1. Introduction

Generators are a special category of processors that have no XML data inputs, only outputs. They are generally used at the top of an XML pipeline to generate XML data from a Java object or other non-XML source. OPS provides several generators as described below. See also the [URL](#) and [Request](#) generators.

4.3.2. Scope Generator

The Scope generator can retrieve documents from the application, session and request scopes. It can work together with the [Scope serializer](#), or retrieve documents stored by custom processors or other application modules.

The following Java object types are supported and checked in this order:

- **org.dom4j.Document** - An XML document represented using the dom4j class hierarchy.
- **org.w3c.dom.Document** - An XML document represented using the W3C DOM class hierarchy.
- **java.lang.String** - An XML document represented as a String.
- **java.lang.Object** - Any Java object. In this case, the object is serialized to XML using [Castor](#). An optional `mapping` input may specify a custom [Castor mapping document](#).

Type	Name	Purpose	Mandatory
Input	config	Configuration	Yes
Input	mapping	Castor XML mapping	No
Output	data	Result XML data	Yes

The `config` input has the following format:

```

<config>
  <key>cart</key>
  <scope>application|session|request</scope>
  <session-scope>application|portlet</session-scope>
</config>

```

key	The <code><key></code> element contains a string used to identify the document. The same key must be used to store and retrieve a document.
scope	<p>The <code><scope></code> element specifies in what scope the document is to be retrieved from. The available scopes are:</p> <ul style="list-style-type: none"> ■ application - The application scope starts when the Web application is deployed. It ends when the Web application is undeployed. The application scope provides an efficient storage for data that does not need to be persisted and that is common for all users. It is typically used to cache information (e.g. configuration data for the application read from a database). ■ session - The session scope is attached to a given user of the Web application. It is typically used to store information that does not need to be persisted and is specific to a given user. It is typically used to cache the user's profile. ■ request - The request scope starts when an HTTP request is sent to the server. It ends when the corresponding HTTP response is sent back to the client. The request scope can be used to integrate a OPS application with legacy J2EE servlets.
session-scope	<p>The <code><session-scope></code> element specifies in what session scope the document is to be retrieved from. This element is only allowed when the <code><scope></code> element is set to <code>session</code>. The available session scopes are:</p> <ul style="list-style-type: none"> ■ application - access the entire application session. This is always a valid value. ■ portlet - access the local portlet session. This is only valid if the processor is run within a portlet. <p>If the element is missing, a default value is used: <code>application</code> when the processor runs within a servlet, and <code>portlet</code> when the processor runs within a portlet.</p>

The optional `mapping` input contains a custom [Castor mapping document](#). This mapping is used only if the object retrieved can only be handled as `java.lang.Object` and not as one of the other types.

The data output contains the document retrieved. When the Scope generator cannot find any document in scope for the given key, it returns a "null document":

```
<null xsi:nil="true"/>
```

Note

The Session generator, previously used to retrieve documents from the session scope, is now deprecated. Use the Scope generator with session scope instead.

Note

The Bean generator, previously used to retrieve JavaBeans from the request and session scopes, is now deprecated. Use the Scope generator instead.

4.3.3. Servlet Include Generator

The Servlet Include generator, using the [RequestDispatcher](#), calls and parses the result of another servlet running in the same Java virtual machine. The servlet can generate either XML or HTML. The generator automatically detects HTML and uses HTMLTidy to clean and parse the stream as XML.

Note

This generator works only in a servlet environment. It is not supported in portlets.

The `config` input describes the servlet to call, and optionally configures the HTMLTidy process. You can specify the servlet either by name or path, and optionally specify a context path. The RelaxNG schema for this input is the following:

```
<element name="config" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <interleave>
    <choice>
      <element name="servlet-name">
        <data type="string"/>
      </element>
      <element name="path">
        <data type="string"/>
      </element>
    </choice>
    <optional>
      <element name="context-uri path">
        <data type="string"/>
      </element>
    </optional>
    <optional>
      <element name="tidy-options">
        <interleave>
          <optional>
            <element name="show-warnings">
              <choice>
                <value>true</value>
                <value>false</value>
              </choice>
            </element>
            <optional>
              <element name="quiet">
                <choice>
                  <value>true</value>
                  <value>false</value>
                </choice>
              </element>
            </optional>
          </optional>
        </interleave>
      </element>
    </optional>
  </interleave>
</element>
```

The data output contains the result in the servlet include call.

This generator calls the `reports` servlet in the `/admin` context path. Since this servlet is generating HTML, it's better to have verbose error reporting from HTMLTidy.

```

<p: processor name="oxf: servlet-include" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <config>
      <servlet-name>reports</servlet-name>
      <context-path>/admin</context-path>
      <tidy-options>
        <show-warnings>true</show-warnings>
        <quiet>false</quiet>
      </tidy-options>
    </config>
  </p: input>
  <p: output name="data" id="#report"/>
</p: processor>

```

4.3.4. Exception Generator

The Exception generator is usually used in an [error pipeline](#). It serializes to XML the data contained in a Java exception retrieved from the request scope, with the following information:

- All the exceptions, starting from the top-level exception down to the root cause.
- For each exception, the exception class name, message, hierarchy of location information, and list of stack trace elements.

The following is an XML document resulting from an exception:

```

<exceptions>
  <exception>
    <type>org.orbeon.oxf.common.Vali dati onExcepti on</type>
    <message>
      null, line 8, column 109, description xforms:bind element: Bind element is missing nodeset attribute null,
      line 8, column 109: Bind element is missing nodeset attribute
    </message>
    <location>
      <system-id/>
      <line>8</line>
      <column>109</column>
      <description>xforms:bind element</description>
      <element>
        <xforms:bind xmlns:xforms="http://www.w3.org/2002/xforms" id="xforms-element-2" ref="/instance/a"
        calculate=". + 1"/>
      </element>
    </location>
    <location>
      <system-id>oxf:/ops/pfc/xforms-epilogue.xpl</system-id>
      <line>99</line>
      <column>66</column>
      <description>reading processor output</description>
      <parameters>
        <parameter>
          <name>name</name>
          <value>response</value>
        </parameter>
        <parameter>
          <name>id</name>
          <value>response</value>
        </parameter>
        <parameter>
          <name>ref</name>
          <value/>
        </parameter>
      </parameters>
      <element>
        <p: output xmlns:p="http://www.orbeon.com/oxf/pipeline" name="response" id="response"/>
      </element>
    </location>
    ...
  </location>
  ...
  <stack-trace-elements>
    <element>
      <class-name>org.orbeon.oxf.common.Vali dati onExcepti on</class-name>
      <method-name>wrapExcepti on</method-name>
      <file-name>Vali dati onExcepti on.java</file-name>
      <line-number>119</line-number>
    </element>
    ...
  </stack-trace-elements>
</exception>
<exception>...</exception>
</exceptions>

```

A typical error pipeline should include an Exception generator followed by one or more transformation(s), and an HTML serializer.

```
<p:processor name="oxf:exception" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:output name="data" id="exception"/>
</p:processor>
```

4.3.5. Bean Generator

Note

This processor is deprecated. Please refer to the [Scope generator](#) instead.

The Bean generator serializes objects in the request or session to XML. If an object is a W3C Document (`org.w3c.dom.Document`), the XML for this document is sent as-is. Otherwise, the object is assumed to be a JavaBean and is serialized to XML with [Castor](#). This generator takes two inputs:

The configuration input describes which bean to serialize, and two optional sources. The sources can be `request`, `session` or both. If both are specified, they are tried in order. In the example below, the request is searched for the `guess` bean. If not found, the session is tried.

```
<p:input name="config" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <config>
    <attribute>guess</attribute>
    <source>request</source>
    <source>session</source>
  </config>
</p:input>
```

Here is the RelaxNG schema:

```
<element name="config" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <interleave>
    <oneOrMore>
      <element name="attribute">
        <data type="string"/>
      </element>
    </oneOrMore>
    <interleave>
      <optional>
        <element name="source">
          <value>request</value>
        </element>
      </optional>
      <optional>
        <element name="source">
          <value>session</value>
        </element>
      </optional>
    </interleave>
  </interleave>
</element>
```

This input specifies the Castor mapping file. See [Castor Documentation](#) for more information on the mapping file. In most instances, the default mapping is sufficient:

```
<p:input name="mapping" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <mapping/>
</p:input>
```

The output document contains a `beans` root node. Each serialized bean is a child of the root node. The example above generates the following document:

```
<beans>
  <guess useraguess="0">
    <message>Hello</message>
  </guess>
</beans>
```

4.4. HTTP Serializer

4.4.1. Scope

Serializers are processors with no XML output. A serializer, just like any processor, can access files, connect to databases, and take actions depending on its inputs.

The HTTP serializer supports decoding binary or text data encapsulated in XML documents and sending it into an HTTP response. Typically, this means sending data back to a client web browser. This can be done in a Servlet environment or a Portlet environment.

4.4.2. Configuration

The configuration consists of the following optional elements.

Element Name	Type	Purpose	Default Value
status-code	valid HTTP status code	HTTP status code sent to the client	SC_OK (HTTP code 100)
error-code	valid HTTP error code	HTTP error code sent to the client	<i>none</i>
content-type	content type, without any attributes	Indicates the content type to send to the client (see decision algorithm below).	<code>application/octet-stream</code> for binary mode, <code>text/plain</code> for text mode
force-content-type	boolean	Indicates whether the content type provided has precedence (see decision algorithm below). This requires a <code>content-type</code> element.	false
ignore-document-content-type	boolean	Indicates whether the content type provided by the input document should be ignored (see decision algorithm below).	false
encoding	valid encoding name	Indicates the text encoding to send to the client (see decision algorithm below).	utf-8
force-encoding	boolean	Indicates whether the encoding provided has precedence (see decision algorithm below). This requires an <code>encoding</code> element.	false
ignore-document-encoding	boolean	Indicates whether the encoding provided by the input document should be ignored (see decision algorithm below).	false
header		Adds a custom HTTP header to the response. The nested elements <code>name</code> and <code>value</code> contain the name and value of the header, respectively. You can add multiple headers.	<i>none</i>
cache-control/ use-local-cache	boolean	Whether the resulting stream must be locally cached. For documents or binaries that are large or known to change at every request, it is recommended to set this to false.	true
empty-content	boolean	Forces the serializer to return an empty content, without reading its data input.	false

Example:

```
<confi g>
  <content-type>i mage/j peg</content-type>
  <header>
    <name>Content-Di sposi ti on</name>
    <val ue>attachment; fi l ename=i mage. j pg</val ue>
  </header>
</confi g>
```

4.4.3. Content Type

The content type sent to the HTTP output is determined according to the following priorities:

1. Use the content type in the `content-type` element of the configuration if `force-content-type` is set to true.
2. Use the content type set in the input document with the `content-type` attribute, if any, unless the `ignore-document-content-type` element of the configuration is set to true.
3. Use the content type in the `content-type` element of the configuration, if specified.
4. Use `application/octet-stream` in binary mode, or `text/plain` in text mode.

4.4.4. Binary Mode

The binary mode is enabled when the root element of the input document contains an `xsi:type` attribute containing a reference to the `xs:base64Encoding` type.

In this mode, all the character content under the root element is considered as Base64-encoded binary content according to the [binary document format](#), for example:

```
<document xsi : type="xs: base64Bi nary" content-type="i mage/j peg">
  /9j /4AAQSkZJRgABAQEBygHKAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwsHBwYGBw4KCggLEA4R ...
  KKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooA/2Q==
</document>
```

An optional `content-type` attribute provides information about the content type of the document. This attribute may be used to set the HTTP `content-type` header, as discussed above.

Here is an example connecting the [URL generator](#) to the serializer, and the appropriate configurations:

```
<p:processor name="oxf:url-generator" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <!-- URL of the image file -->
      <url>oxf:/my-image.jpg</url>
      <!-- Set content type -->
      <content-type>image/jpeg</content-type>
      <!-- This makes sure that the content type specified is used and that the file is read in binary mode,
      even if the connection sets another content type. With the oxf: protocol, this is not strictly necessary,
      but if the http: protocol was used, this could be used to override the content type set by the HTTP
      connection. -->
      <force-content-type>true</force-content-type>
    </config>
  </p:input>
  <p:output name="data" id="image-data"/>
</p:processor>
<p:processor name="oxf:http-serializer" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <!-- This makes sure that the correct content type is sent to the client. In this particular case, this
      is not strictly necessary, as the content type provided by the input document would be used anyway. -->
      <content-type>image/jpeg</content-type>
      <force-content-type>true</force-content-type>
    </config>
  </p:input>
  <p:input name="data" href="#image-data"/>
</p:processor>
```

4.4.5. Text Mode

The text mode is enabled when the root element of the input document contains an `xml:content-type` attribute containing a reference to the `xs:string` type.

In this mode, all the character content under the root element is considered as text content according to the [text document format](#), for example:

```
<document xml:content-type="xs:string" content-type="text/plain">
  Rien n'est beau comme ces maisons du siècle dix-septième dont la place Royale offre une si majestueuse réunion.
  Quand leurs faces de briques, entremêlées et encadrées de cordons et de coins de pierre, et quand leurs fenêtres
  hautes sont enflammées des rayons splendides du couchant, vous vous sentez, à les voir, la même vénération que
  devant une Cour des parlements assemblée en robes rouges à revers d'hermine ; et, si ce n'était un puéril
  rapprochement, on pourrait dire que la longue table verte où ces redoutables magistrats sont rangés en carré
  figure un peu ce bandeau de tilleuls qui borde les quatre faces de la place Royale et en complète la grave
  harmonie.
</document>
```

An optional `content-type` attribute provides information about the content type of the document. This attribute may be used to set the HTTP `content-type` header, as discussed above. In text mode, the `content-type` attribute can also have a `charset` parameter providing a hint at the preferred character encoding for the text, as discussed below, for example `text/plain; charset=iso-8859-1`. Note that the XML input document internally represents characters in Unicode and therefore does not require encoding information.

The character encoding sent to the HTTP output is determined according to the following priorities:

1. Use the encoding in the `encoding` element of the configuration if `force-encoding` is set to `true`.
2. Use the encoding set in the input document with the `content-type` attribute, if any, unless the `ignore-document-encoding` element of the configuration is set to `true`.
3. Use the encoding in the `encoding` element of the configuration, if specified.
4. Use `utf-8`.

[RFC 2183](#) describes the `Content-Disposition` HTTP header, used by web browsers to decide how to display an attachment. A value of `inline` means that a browser will try to use a plugin to display for example a PDF file. A value of `attachment` causes the browser to ask the user to save the file, optionally proposing a file name. The following two examples show how it is possible to specify such headers with the HTTP serializer:

Using a plugin:

```
<p:processor name="oxf:xsl fo-serializer" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <header>
        <name>Content-Disposition</name>
        <value>inline</value>
      </header>
    </config>
  </p:input>
  <p:output name="data" href="#image-data"/>
</p:processor>
```

```

</p:input>
<p:input name="data" href="#pdf-document"/>
</p:processor>

```

Saving a file to disk:

```

<p:processor name="oxf:xml-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <header>
        <name>Content-Disposition</name>
        <value>attachment; filename=report.pdf</value>
      </header>
    </config>
  </p:input>
  <p:input name="data" href="#pdf-document"/>
</p:processor>

```

Here is an example connecting the [URL generator](#) to the serializer, and the appropriate configurations:

```

<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <!-- URL of the text file -->
      <url>oxf:/text.txt</url>
      <!-- The file will be read as plain text -->
      <content-type>text/plain</content-type>
      <!-- The file is encoded with this encoding -->
      <encoding>iso-8859-1</encoding>
    </config>
  </p:input>
  <p:output name="data" id="text-data"/>
</p:processor>
<p:processor name="oxf:http-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <!-- Make sure the client receives the text/plain content type -->
      <content-type>text/plain</content-type>
      <force-content-type>true</force-content-type>
      <!-- We specify another encoding, and force it -->
      <encoding>utf-8</encoding>
      <force-encoding>true</force-encoding>
    </config>
  </p:input>
  <p:input name="data" href="#text-data"/>
</p:processor>

```

Note

The HTTP serializer performs streaming. If its input data consists of a stream of short character SAX events, such as those produced by the [Request generator](#) the [URL generator](#), the [SQL processor](#), or converter processors, it is possible to serialize an "infinitely" long document.

Note

When using the command-line mode, instead of sending the output through HTTP, the HTTP serializer sends its output to the standard console output. In such a case, the parameters that do not affect the content of the data, such as content-type, status-code, etc. are ignored.

Note

The HTTP serializer sends the cache control HTTP headers, including Last-Modified, Expires and Cache-Control. The Content-Type and Content-Length headers are also supported.

4.5. Other Serializers

4.5.1. Scope

Serializers are processors with no XML output. A serializer, just like any processor, can access files, connect to databases, and take actions depending on its inputs. See also the [HTTP serializer](#).

4.5.2. URL Serializer

The URL Serializer mirrors the functionality of the [URL Generator](#). Instead of reading from of URL, it writes its data input as XML in a URL. Note that only the `oxf:` and `http:` protocols allows writing at this time.

Note

The `Filesystem` resource manager is the only Resource Manager supporting write operations.

The URL serializer takes a `config` input with a single `url` element containing the URL to write to. The data input is serialized according the rules of the XML Serializer.

```
<p:processor name="oxf:url-serializer" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>oxf:/path/current.xml</url>
    </config>
  </p:input>
  <p:input name="data" href="#xml-data"/>
</p:processor>
```

4.5.3. File Serializer

The File Serializer is most useful in a `command-line` application. When executed, this serializer writes its data input into the file specified in the `config` element. Here is an example:

```
<p:processor name="oxf:file-serializer" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <file>test.html</file>
      <directory>c:/build/doc</directory>
      <content-type>text/html</content-type>
      <public-doctype>-//W3C//DTD HTML 4.01//EN</public-doctype>
      <system-doctype>http://www.w3.org/TR/html4/strict.dtd</system-doctype>
      <encoding>utf-8</encoding>
      <indent-amount>4</indent-amount>
    </config>
  </p:input>
  <p:input name="data" href="#html"/>
</p:processor>
```

The `file` element specifies the file to write to.

The `directory` element is optional. If specified, it is used as a base directory for the `file` element.

The `content-type` element is optional. The default is `application/xml`. Other valid values are `text/html` and `text/plain`.

The following optional attributes can also be specified: `public-doctype`, `system-doctype`, `encoding`, `indent` and `indent-amount`.

4.5.4. Scope Serializer

The Scope serializer can store documents into the application, session and request scopes. It works together with the [Scope generator](#).

The Scope serializer has a `config` input in the following format:

```
<config>
  <key>cart</key>
  <scope>application|session|request</scope>
  <session-scope>application|portlet</session-scope>
</config>
```

key	The <code><key></code> element contains a string used to identify the document. The same key must be used to store and retrieve a document.
scope	<p>The <code><scope></code> element specifies in what scope the document is to be stored. The available scopes are:</p> <ul style="list-style-type: none">■ application - The application scope starts when the Web application is deployed. It ends when the Web application is undeployed. The application scope provides an efficient storage for data that does not need to be persisted and that is common for all users. It is typically used to cache information (e.g. configuration data for the application read from a database).■ session - The session scope is attached to a given user of the Web application. It is typically used to store information that does not need to be persisted and is specific to a given user. It is typically used to cache the user's profile.■ request - The request scope starts when an HTTP request is sent to the server. It ends when the corresponding HTTP response is sent back to the client. The request scope can be used to integrate a OPS application with legacy J2EE servlets.
	<p>The <code><session-scope></code> element specifies in what session scope the document is to be stored. This element is only allowed when the <code><scope></code> element is set to <code>session</code>. The available session scopes are:</p> <ul style="list-style-type: none">■ application - access the entire application session. This is always a valid value.■ portlet - access the local portlet session. This is only valid if the processor is run within a portlet.

If the element is missing, a default value is used: `application` when the processor runs within a servlet, and `portlet` when the processor runs within a portlet.

In addition to the `config` input, the Scope serializer has a `data` input receiving the document to store.

Note

The Session serializer, previously used, is now deprecated. Use the Scope serializer with session scope instead.

4.5.5. Null Serializer

The Null Serializer acts as a black hole. The `data` input is read and ignored. This processor is useful when a pipeline or a branch of a `p:choose` element doesn't have to return any document.

```
<p:processor name="oxf:null-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data" href="#document"/>
</p:processor>
```

4.5.6. Flushing the Output Stream

All serializers (XML, HTML, text, and FOP) will flush their output stream when they encounter the following processing instruction: `<?oxf-serializer flush?>`

This instruction allows the browser to display a Web page incrementally. Incremental display is typically useful when sending large tables or when the first part of a Web page could be displayed, while the rest of the page cannot until a time consuming action is performed.

4.5.7. Legacy HTTP Serializers

Note

Use of these serializers should be replaced by `converters` connected to the `HTTP serializer`.

These serializers share a common functionality: writing their data input to an HTTP response. Typically, this means sending data back to a client web browser. This can be done in a Servlet environment or a Portlet environment. All share the same configuration, but differ in how they convert their input data. The following describes the common configuration, then the specifics for each serializer.

Note

When using the command-line mode, instead of sending the output through HTTP, the HTTP serializers send their output to the standard output. In such a case, the parameters that do not affect the content of the data, such as `content-type`, `status-code`, etc. are ignored.

All serializers send the cache control HTTP headers, including `Last-Modified`, `Expires` and `Cache-Control`. The `content-type` and `content-length` headers are also supported.

The configuration consists of the following optional elements.

Element	Purpose	Default
<code>content-type</code>	content type sent to the client	Specific to each serializer
<code>encoding</code>	The default text encoding	utf-8
<code>status-code</code>	HTTP status code sent to the client	SC_OK, or 100
<code>error-code</code>	HTTP error code sent to the client	none
<code>empty-content</code>	Forces the serializer to return an empty content, without reading its data input	false
<code>version</code>	HTML or XML version number	4.01 for HTML (ignored for XML, which always output 1.0)
<code>public-doctype</code>	The public doctype	"-//W3C//DTD HTML 4.01 Transitional//EN" for HTML, none otherwise
<code>system-doctype</code>	The system doctype	"http://www.w3.org/TR/html4/loose.dtd" for HTML, none otherwise
<code>omit-xml-declaration</code>	Specifies whether an XML declaration must be omitted	false for XML and HTML (i.e. a declaration is output by default), ignored otherwise
<code>standalone</code>	If true, specifies <code>standalone="yes"</code> in the document declaration. If false, specifies <code>standalone="no"</code> in the document declaration.	not specified for XML, ignored otherwise

indent	Specifies if the output is indented	true
indent-amount	Specifies the number of indentation space	1
cache-control/use-local-cache	Whether the resulting stream must be locally cached. For documents or binaries that are large or known to change at every request, it is recommended to set this to false.	true
header	Adds a custom HTTP header to the response. The nested elements <code>name</code> and <code>value</code> contain the name and value of the header, respectively. You can add multiple headers.	none

```
<config>
  <content-type>text/html </content-type>
  <status-code>100</status-code>
  <empty-content>>false</empty-content>
  <error-code>0</error-code>
  <version>4.01</version>
  <public-doctype>-//W3C//DTD HTML 4.01//EN</public-doctype>
  <system-doctype>http://www.w3.org/TR/html4/strict.dtd</system-doctype>
  <omit-xml-declaration>>false</omit-xml-declaration>
  <standalone>true</standalone>
  <encoding>utf-8</encoding>
  <indent-amount>4</indent-amount>
  <cache-control>
    <use-local-cache>true</use-local-cache>
  </cache-control>
  <header>
    <name>Content-Disposition</name>
    <value>attachment; filename=image.jpeg</value>
  </header>
</config>
```

This serializer writes XML text. The output is indented with no spaces and encoded using the UTF-8 character set. The default content type is `application/xml`.

```
<p:processor name="oxf:xml-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <content-type>text/vnd.wap.wml</content-type>
    </config>
  </p:input>
  <p:input name="data" href="#wml"/>
</p:processor>
```

The HTML Serializer's output conforms to the XSLT `html` semantic. The `doctype` is set to `HTML 4.0 Transitional` and the content is indented with no space and encoded using the UTF-8 character set. The default content type is `text/html`. The following is a simple `HTMLSerializer` example:

```
<p:processor name="oxf:html-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config/>
  </p:input>
  <p:input name="data" href="#html"/>
</p:processor>
```

Note

The XML 1.0 Specification prohibits a DOCTYPE definition with a Public ID and no System ID.

The Text Serializer's output conforms to the XSLT `text` semantic. The output is encoded using the UTF-8 character set. This serializer is typically useful for pipelines generating Comma Separated Value (CSV) files. The default content type is `text/plain`.

```
<p:processor name="oxf:text-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config/>
  </p:input>
  <p:input name="data" href="#text"/>
</p:processor>
```

4.6. Converters

4.6.1. Introduction

Converters are processors converting XML documents from one format to another. For example, the standard HTML converter documented below converts an XML document into an HTML document. This HTML document can then be sent to a web browser using the [HTTP serializer](#), or attached to an email with the [Email processor](#).

Converters typically have a `data` output containing the converted document.

4.6.2. Standard Converters

The standard converters convert XML infosets (the XML documents that circulate in OPS pipelines) into text according to standard output methods defined by the XSLT specification. They convert to the following formats:

- **XML**: a standard XML document
- **HTML**: a standard HTML document
- **XHTML**: a standard XHTML document
- **Text**: any text document

The resulting text is sent to the `data` output. It is embedded in an XML document as specified by the [text document format](#).

The configuration of the standard converters consists of the following optional elements:

Element	Purpose	Default
method	XSLT output method (one of <code>xml</code> , <code>html</code> , <code>xhtml</code> or <code>text</code>)	<code>xml</code> , <code>html</code> or <code>text</code> , depending on the serializer
content-type	Content type hint specified on the output <code>document</code> element	Specific to each serializer
encoding	Encoding hint specified on the output <code>document</code> element	<code>utf-8</code>
version	HTML or XML version number	4.01 for HTML (ignored for XML, which always output 1.0)
public-doctype	The public doctype	" <code>http://www.w3.org/TR/html4/loose.dtd</code> " for HTML, none otherwise
system-doctype	The system doctype	" <code>http://www.w3.org/TR/html4/loose.dtd</code> " for HTML, none otherwise
omit-xml-declaration	Specifies whether an XML declaration must be omitted	false for XML and HTML (i.e. a declaration is output by default), ignored otherwise
standalone	If true, specifies <code>standalone="yes"</code> in the document declaration. If false, specifies <code>standalone="no"</code> in the document declaration. If missing, no <code>standalone</code> attribute is produced. For more information about standalone document declarations, please refer to the relevant section of the XML specification . In most cases, this does not need to be specified.	not specified for XML, ignored otherwise
indent	Specifies if the output is indented. This means that line breaks maybe be inserted between adjacent elements. The actual level of indentation is specified with the <code>indent-amount</code> configuration element.	true (ignored for text method)
indent-amount	Specifies the number of indentation space	1 (ignored for text method)

Example:

```
<config>
  <content-type>text/html </content-type>
  <encoding>utf-8</encoding>
  <version>4.01</version>
  <public-doctype>-//W3C//DTD HTML 4.01//EN</public-doctype>
  <system-doctype>http://www.w3.org/TR/html4/strict.dtd</system-doctype>
  <indent-amount>4</indent-amount>
</config>
```

The XML converter outputs an XML document conform to the XSLT `xml` semantic. By default, the output is indented with no spaces and encoded using the UTF-8 character set. The default MIME content type is `application/xml`. The following is a simple XML converter example:

```
<p:processor name="oxf:xml-converter" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <content-type>application/xml </content-type>
      <encoding>iso-8859-1</encoding>
      <version>1.0</version>
```

```

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

```

This is an example of output produced by the XML converter:

```

<document xsi:type="xs:string" content-type="application/xml" charset="iso-8859-1">
  <?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
  <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/>
        </name-info>
        <address>
          <address-detail>
            <street-name>N Columbus Dr.</street-name>
            <street-number>511</street-number>
            <unit-number/>
          </address-detail>
            <city>Chicago</city>
            <state-province>IL</state-province>
            <postal-code>60611</postal-code>
            <country>USA</country>
            <email>j.doe@acme.org</email>
          </address>
        </general-info>
        <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>
        <family-info>
          <children>
            <child>
              <birth-date>2003-02-02</birth-date>
              <first-name>Marco</first-name>
            </child>
            <child>
              <birth-date/>
              <first-name/>
            </child>
          </children>
          <comments>No comments at this point!</comments>
          <family-info>
            <claim-info>
              <accident-type>FOOT</accident-type>
              <accident-date>2004-07-06</accident-date>
              <rate/>
            </claim-info>
          </family-info>
        </insured-info>
      </claim>
    </document>

```

The HTML converter outputs an HTML document conform to the XSLT `html` semantic. By default, the doctype is set to `HTML 4.0 Transitional` and the content is indented with no space and encoded using the UTF-8 character set. The default content type is `text/html`. The following is a simple HTML converter example:

```

<p:processor name="oxf:html-converter" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <content-type>text/html</content-type>
      <encoding>iso-8859-1</encoding>
      <public-doctype>-//W3C//DTD HTML 4.01 Transitional//EN</public-doctype>
      <version>4.01</version>
    </config>
  </p:input>
  <p:input name="data">
    <html>
      <head>
        <title>My HTML document</title>
      </head>
      <body>
        <p>
          This is the content of the HTML document.
        </p>
      </body>
    </html>
  </p:input>
  <p:output name="data" id="html-document"/>
</p:processor>

```

This is an example of output produced by the HTML converter:

```

<document xsi:type="xs:string" content-type="text/html" charset="iso-8859-1">
  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
  <html>
    <head>
      <title>My HTML document</title>
    </head>
    <body>
      <p> This is the content of the HTML document. </p>
    </body>
  </html>
</document>

```

Note

The XML 1.0 Specification prohibits a DOCTYPE definition with a Public ID and no System ID.

The Text converter outputs a text document conform to the XSLT `text` semantic. By default, the output is encoded using the UTF-8 character set. This serializer is typically useful for pipelines generating Comma Separated Value (CSV) files. The default content type is `text/plain`. The following is a simple Text converter example:

```

<p:processor name="oxf:text-converter" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config/>
  </p:input>
  <p:input name="data">
    <document>
      This is just plain text. It will be output without the<em>text</em>and<em>em</em>elements.
    </document>
  </p:input>
  <p:output name="data" id="text-document"/>
</p:processor>

```

This is an example of output produced by the Text converter:


```
<document xsi:type="xs:string" content-type="text/plain; charset=utf-8">
  This is just plain text. It will be output without the text and em elements.
</document>
```

4.6.3. To-XML Converter

The To-XML Converter produces parsed XML documents from a binary document format.

The data input of the To-XML Converter follows the [binary document format](#). Its data output is an XML document. The mandatory `config` input contains an empty `config` element reserved for future configuration parameters. This is an example of use:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:param type="output" name="data"/>
  <p:processor name="oxf:url-generator">
    <p:input name="config">
      <config>
        <url>paring-vi ew.xsl </url>
        <content-type>binary/octet-stream</content-type>
        <force-content-type>true</force-content-type>
      </config>
    </p:input>
    <p:output name="data" id="xml-file-as-binary"/>
  </p:processor>
  <p:processor name="oxf:to-xml-converter">
    <p:input name="data" href="#xml-file-as-binary"/>
    <p:input name="config">
      <config/>
    </p:input>
    <p:output name="data" ref="data"/>
  </p:processor>
</p:config>
```

4.6.4. XSL-FO Converter

The XSL-FO Converter produces PDF documents from an [XSL-FO](#) description of the page. The default content type is `application/pdf`.

Note

The input document of the XSL-FO must follow the W3C XSL/FO Recommendation . Note that only subset of the recommendation implemented by FOP 0.20.5 is supported.

The resulting binary stream is sent to the data output. It is embedded in an XML document as specified by the [binary document format](#).

4.6.5. XLS Converters

OPS ships with the [POI](#) library which allows import and export of Microsoft Excel documents. OPS uses an Excel file template to define the layout of the spreadsheet. You define cells that will contain the values with a special markup.

First, create an Excel spreadsheet with the formatting of your choosing. Apply a special markup to the cell you need to export values to:

1. Select the cell
2. Go to the menu `Format->Cell`
3. In the `Number` tab, choose the `Custom` format and enter a format that looks like: `#,##0;" /a/b| /c/d"`. In this example we have 2 XPath expressions separated by a pipe character (`|`): `/a/b` and `/c/d`. The first XPath expression is used when creating the Excel file (exporting) and is run against the data input document of the To XLS converter. The second expression is optional and is used when recreating an XML document from the Excel file (importing with the From XLS converter).

The To XLS converter takes a `config` input describing the XLS template file, and a `data` input containing the values to be inserted in the template. The processor scans the template, and applies XPath expressions to fill in the template. It returns a [binary document](#) on its data output.

The `config` input takes a single `config` element with one attribute:

template	A URL pointing to an XLS template file
-----------------	--

```
<p:processor name="oxf:xls-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config template="oxf:/excel/template.xls"/>
  </p:input>
  <p:input name="data">
    <currency>
      <val ue1>10</val ue1>
      <val ue2>20</val ue2>
      <val ue3>30</val ue3>
    </currency>
  </p:input>
</p:processor>
```

```

<p:processor name="oxf:to-xls-converter" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config template="oxf:/examples/employees/export-excel/employees.xls">
      <repeat-row row-num="3" for-each="employees/employee"/>
    </config>
  </p:input>
  <p:input name="data" href="#workbook"/>
  <p:output name="data" id="xls-binary"/>
</p:processor>
<!-- Serialize -->
<p:processor name="oxf:http-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data" href="#xls-binary"/>
  <p:input name="config">
    <config>
      <header>
        <name>Content-Disposition</name>
        <value>attachment; filename=employees.xls</value>
      </header>
    </config>
  </p:input>
</p:processor>

```

The From XLS converter takes an Excel file (for example uploaded with an XForms upload control), finds special markup cells and reconstructs an XML document from this markup. The converter has one data input which must receive a [binary document](#), and a data output containing the generated XML document. Assume the following XForms model:

```

<xf:model xmlns:xf="http://www.w3.org/2002/xforms">
  <xf:instance>
    <form>
      <action/>
      <files>
        <file filename="" mediatype="" size="" xsi:type="xs:anyURI"/>
      </files>
    </form>
  </xf:instance>
  <xf:submission method="post" encoding="multipart/form-data"/>
</xf:model>

```

The model can be filled with the following XForms controls:

```

<xforms:group ref="/form" xmlns:xforms="http://www.w3.org/2002/xforms">
  <p>
    <xforms:upload ref="files/file[1]"/>
    <xforms:submit>
      <xforms:label>Submit</xforms:label>
      <xforms:setvalue ref="action">import</xforms:setvalue>
    </xforms:submit>
  </p>
</xforms:group>

```

Then the following pipeline can extract the data from the uploaded file:

```

<!-- Dereference URI stored in instance and return a binary -->
<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config" href="aggregate('config', aggregate('url', #instance#xpointer(string(/form/files/file[1])), aggregate('content-type', #instance#xpointer('application/octet-stream')))" />
  <p:output name="data" id="xls-binary"/>
</p:processor>
<!-- Convert file to XML -->
<p:processor name="oxf:from-xls-converter" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data" href="#xls-binary"/>
  <p:output name="data" id="xls"/>
</p:processor>

```

This is an example of returned document, given an appropriate configuration of the Excel template:

```

<workbook>
  <sheet>
    <employees>
      <employee-id>5398</employee-id>
      <firstname>Niels</firstname>
      <lastname>Aas</lastname>
      <phone>(555) 123 0434</phone>
      <title>Norwegian sculptor and illustrator</title>
      <age>70</age>
      <manager-id/>
      <employee-id>5028</employee-id>
      <firstname>Ali</firstname>
      <lastname>Abbasi</lastname>
      <phone>(555) 123 0060</phone>
    </employees>
  </sheet>
</workbook>

```

```

<title>BBC Scotland travel presenter</title>
<age>42</age>
<manager-id/>
</employees>
</sheet>
</workbook>

```

4.7. XSLT and JAXP Processors

4.7.1. Introduction

XSLT 1.0 and XSLT 2.0 are languages for transforming XML documents into other XML documents. OPS uses XSLT extensively to process XML documents.

OPS ships with multiple implementations of XSLT:

- Saxon 8.1.1 (supports XSLT 2.0 and XPath 2.0 in addition to XSLT 1.0 and XPath 1.0)
- Saxon 6.5.2 (supports XSLT 1.0)
- Xalan 2.5.1 Interpreter (supports XSLT 1.0)
- Xalan 2.5.1 Compiler (XSLTC) (supports XSLT 1.0)

Additionally, OPS can use any Java API for XML Processing (JAXP) compliant transformer.

4.7.2. Inputs and Outputs

Type	Name	Purpose	Mandatory
Input	transformer	Specifies a concrete <code>TransformerFactory</code> class name.	Yes Implicit with concrete processors.
Input	attributes	Allows setting JAXP TransformerFactory attributes.	No Implicit with <code>oxf:saxon8</code> , <code>oxf:unsafe-saxon8</code> , <code>oxf:xslt</code> , <code>oxf:unsafe-xslt</code>
Input	config	The description of the transformation, typically an XSLT stylesheet.	Yes
Input	data	The document to which the stylesheet is applied.	Yes
Input	User-defined inputs	User-defined inputs accessible with the XPath <code>document()</code> or <code>doc()</code> functions.	No
Output	data	The result of the transformation.	Yes

4.7.3. Processor Names

There are three generic processors: a generic JAXP (TrAX) processor, and two specialized XSLT processors (one for XSLT 1.0 and one for XSLT 2.0), which provides additional functionality related to XSLT:

Processor Name	Language	Implementation
oxf:generic-xslt-1.0	XSLT 1.0	Generic JAXP (TrAX) XSLT 1.0 transformer (configure with <code>transformer</code> input)
oxf:generic-xslt-2.0	XSLT 2.0	Generic JAXP (TrAX) XSLT 2.0 transformer (configure with <code>transformer</code> input)
oxf:trax	Any	Generic JAXP (TrAX) transformer (configure with <code>transformer</code> input)

You usually don't use directly the generic processors. Instead, you use concrete processors that leverage the generic processors:

Processor Name	Language	Implementation
oxf:xslt	XSLT 1.0 and 2.0	Saxon 8.1.1 with external functions disabled
oxf:unsafe-xslt	XSLT 1.0 and 2.0	Saxon 8.1.1 with external functions enabled
oxf:xalan	XSLT 1.0	Xalan 2.5.1 Interpreter
oxf:xsltc	XSLT 1.0	Xalan 2.5.1 Compiler
oxf:saxon	XSLT 1.0	Saxon 6.5.2
oxf:saxon8	XSLT 1.0 and 2.0	Saxon 8.1.1 with external functions disabled
oxf:unsafe-saxon8	XSLT 1.0 and 2.0	Saxon 8.1.1 with external functions enabled
oxf:xslt-1.0	XSLT 1.0	Default JAXP (TrAX) XSLT 1.0 transformer (currently Saxon 8)
oxf:xslt-2.0	XSLT 2.0	Default JAXP (TrAX) XSLT 2.0 transformer (currently Saxon 8)
oxf:pfc-xslt-1.0	XSLT 1.0	Default JAXP (TrAX) XSLT 1.0 transformer used by the Page Flow Controller (currently Saxon 8)
oxf:pfc-xslt-2.0	XSLT 2.0	Default JAXP (TrAX) XSLT 2.0 transformer used by the Page Flow Controller (currently Saxon 8)

The most commonly used processor is `oxf:xslt`, which provides excellent support for XSLT 1.0 and XSLT 2.0.

4.7.4. Usage

The transformer input is mandatory only when using the generic processor: `oxf:generic-xslt-1.0`, `oxf:generic-xslt-2.0` or `oxf:trax`. It is implied otherwise.

This input specifies a specific concrete subclass of `TransformerFactory`. For example, you can use the TrAX transformer to interface OPS with a new hypothetical transformer (MyTrans):

```
<p: processor name="oxf:trax" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="transformer">
    <config>
      <class>com.mytrans.TransformerFactoryImpl</class>
    </config>
  </p: input>
  <p: input name="config">
    <mytrans>
      ...
    </mytrans>
  </p: input>
  <p: input name="data">
    <document>
      ...
    </document>
  </p: input>
  <p: output name="data" id="mytrans"/>
</p: processor>
```

The `attributes` input is optional and is used to specify JAXP `TransformerFactory` attributes. It starts with a `attributes` element containing a number of `attribute` elements. Each element specifies a name, a type, and a value, all dependent on the actual transformer implementation. Saxon does not support the same attributes as Xalan, for example. The following types are supported:

- `xs:string`: passed as a Java String
- `xs:integer`: passed as a Java Integer
- `xs:boolean`: passed as a Java Boolean
- `xs:date`: passed as a Java Date
- `xs:dateTime`: passed as a Java Date
- `xs:anyURI`: passed as a Java URL

This is an example of configuration supported by Saxon:

```
<attributes>
  <attribute as="xs:boolean" name="http://saxon.sf.net/feature/allow-external-functions" value="false"/>
  <attribute as="xs:boolean" name="http://saxon.sf.net/feature/linenumbering" value="false"/>
  <attribute as="xs:integer" name="http://saxon.sf.net/feature/recoveryPolicy" value="2"/>
</attributes>
```

The `attributes` input is implicit with `oxf:saxon8`, `oxf:unsafe-saxon8`, `oxf:xslt`, `oxf:unsafe-xslt`.

The `config` input contains a transformation, typically an XSLT stylesheet. The following example shows a simple XSLT stylesheet:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <new-root>
      <xsl:value-of select="/root/a"/>
      <xsl:value-of select="/root/b"/>
    </new-root>
  </xsl:template>
</xsl:stylesheet>
```

The `data` input contains the source XML document. The example below works with the stylesheet shown above:

```
<root>
  <a>a</a>
  <b>b</b>
</root>
```

The `data` output produces the result of the transformation. The following XML is the result of the above transformation:

```
<new-root>ab</new-root>
```

XSLT transformers support reading optional user-defined inputs through the XSLT and XPath `document()` and `doc()` functions. For example:

```

<p:processor name="oxf:xslt" xml:ns:p="http://www.orbeon.com/oxf/pipe" >
  <p:input name="config">
    <address-book xsl:version="2.0">
      <xsl:variable name="document-1" select="doc('file:/document-1.xml')" xml:ns:xsl="http://www.w3.org/1999/XSL/Transform"/>
      <xsl:variable name="document-2" select="doc('input:my-input')" xml:ns:xsl="http://www.w3.org/1999/XSL/Transform"/>
      <!-- ... -->
    </address-book>
  </p:input>
  <p:input name="data" href="#some-id"/>
  <p:input name="my-input" href="#some-other-id"/>
  <p:output name="data" id="address-book"/>
</p:processor>

```

In this example, the XPath expression `doc('file:/document-1.xml')` instructs the XSLT transformer to read the document stored at `oxf:/document-1.xml` and make it accessible through the variable named `document-1`. This is standard XSLT and XPath 2.0.

The second variable, `document-2`, is assigned using a similar XPath expression, but it uses a URI with a particular syntax: it starts with the scheme `input:`. This instructs the XSLT transformer to read the processor input called `my-input`. This input is connected to the XSLT processor with `<p:input name="my-input" href="#some-other-id"/>`. In this case, it is up to the user of the XSLT processor to connect additional inputs as she sees fit, as long as their names don't conflict with mandatory input and output names.

4.7.5. Passing Parameters to XSLT Stylesheets

XSLT supports `xsl:param` elements at the top level of a stylesheet. They allow passing parameters to the stylesheet, without modifying the stylesheet itself. With API such as JAXP, a programmer can associate values to those parameters, making them available to the stylesheet during execution.

XPL does not have a particular mechanism to set such stylesheet parameters, but a very simple workaround allows setting them. Assume you have a stylesheet with a `start` parameter, in a file called `my-stylesheet.xml`:

```

<xsl:transform xml:ns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="start" select="'a1'"/>
  <!-- ... -->
</xsl:transform>

```

To set the `start` parameter with XPL, simply encapsulate the `my-stylesheet.xml` stylesheet as follows:

```

<p:processor name="oxf:xslt" xml:ns:p="http://www.orbeon.com/oxf/pipe" >
  <p:input name="data" href="#stylesheet-input"/>
  <p:input name="config">
    <xsl:stylesheet version="2.0" xml:ns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:import href="my-stylesheet.xml"/>
      <xsl:param name="start" select="'a2'"/>
    </xsl:stylesheet>
  </p:input>
  <p:output name="data" id="stylesheet-output"/>
</p:processor>

```

The rules of XSLT's `import` statement will make sure that the imported stylesheet, `my-stylesheet.xml`, receives the `start` parameter with the value "a2", as overridden in the importing stylesheet. Of course, the value does not have to be static, it can be computed dynamically as well.

4.7.6. Streaming Transformations for XML (STX)

Streaming Transformations for XML (STX) is a one-pass transformation language for XML documents that builds on the Simple API for XML (SAX). STX is intended as a high-speed, low memory consumption alternative to XSLT. Since it does not require the construction of an in-memory tree, it is suitable for use in resource constrained scenarios.

OPS uses Joost to implement a STX transformer. Its usage is similar to an XSLT transformer, using the processor URI `oxf:stx`. This example demonstrates a simple STX transformation.

```

<p:processor name="oxf:stx" xml:ns:p="http://www.orbeon.com/oxf/pipe" >
  <p:input name="data" href="#source"/>
  <p:input name="config">
    <stx:transform version="1.0" pass-through="all" strip-space="yes" xml:ns:stx="http://stx.sourceforge.net/2002/ns"></stx:transform>
  </p:input>
  <p:input name="data" href="#document"/>
  <p:output name="data" id="result"/>
</p:processor>

```

4.8. XInclude Processor

4.8.1. Introduction

XInclude 1.0 is a simple language for XML inclusions. It's main use is to specify that content from a target XML document must be included at a certain place within a source XML document.

4.8.2. Note About XInclude in Orbeon PresentationServer

By default, XInclude processing is not automatic in OPS, and must be enabled explicitly, except in a few cases described below.

There are two ways to use XInclude explicitly in OPS:

- **At parsing time.** The XML parser included with OPS is able to process XInclude instructions when an XML file is being parsed (i.e. loaded) into OPS. This is usually controlled with the [URL generator](#). Example:

```
<p:processor name="oxf:url-generator">
  <p:input name="config">
    <config>
      <url>oxf:/some-document.xml</url>
      <handle-xinclude>true</handle-xinclude>
    </config>
  </p:input>
  <p:output name="data" id="processed-document"/>
</p:processor>
```

- **With the XInclude processor.** The processor documented here allows to explicitly control the use of XInclude instructions in any XML document, including documents that are not loaded with the URL generator, but dynamically generated. Example:

```
<p:processor name="oxf:xinclude">
  <p:input name="config" href="oxf:/some-document.xml"/>
  <p:output name="data" id="processed-document"/>
</p:processor>
```

This example is equivalent to the example using the URL generator.

XInclude processing is implicit in the following cases:

- **Page Flow Controller (PFC) Components.** PFC page models, page views, and actions automatically go through XInclude processing. This in particular allows for using XInclude in static XHTML or dynamic XSLT page views. In this case, XInclude can make use of the special URIs `input:data` and `input:instance` to access data and instance documents fed to the page view.

Note

Because the PFC automatically processes XInclude statements in page components, the user should be careful when using XInclude statements in XPL pipelines for such components. It should be understood that XInclude processing in this case takes place upon loading the XPL pipeline.

- **XPL Processor Definitions.** Processor definitions (usually stored in a file called `processors.xml`) may use XInclude.
- **Logging Configuration.** Logging configuration (usually stored in a file called `log4j.xml`) may use XInclude.

It must be noted that, in particular, XInclude processing is **not** implicit in the following cases:

- **XPL href attribute.** In the following example:

```
<p:input name="my-input-name" href="some-document.xml"/>
```

In this case, the content of `some-document.xml` will not be processed by XInclude before being sent to the `my-input-name` input.

- **XPL schema-href attribute.** Similarly, XML schemas do not go through XInclude processing.

4.8.3. Inputs and Outputs

Type	Name	Purpose	Mandatory
Input	config	XML document with XInclude elements.	Yes
Input	User-defined inputs	User-defined inputs accessible with URLs of the form <code>input:custom-input</code> .	No
Output	data	The result of the transformation.	Yes

4.8.4. Usage

The `config` input receives an XML document which may contain XInclude elements. If it doesn't contain any XInclude elements, the content of the document will be produced without modifications on the XInclude processor's `data` output.

The main XInclude element is `xi:include`. The prefix, usually `xi`, must be bound to the namespace `http://www.w3.org/2001/XInclude`. `xi:include` takes a mandatory `href` attribute, which must contain an URL such as an `oxf:`, `file:` or `http:` URL. In addition, URIs of the form `input:*` are supported, to allow access to user-defined inputs connected to the XInclude processor.

This is a fragment of an XForms page using XInclude, in a resource named `view.xhtml`:

```
<xhtml:html>
  <xhtml:head>
    <xhtml:title>XForms Text Controls</xhtml:title>
    <xforms:model id="main-model">
      <xforms:instance id="instance">
        <!-- This is where the inclusion is performed -->
        <xi:include href="main-xforms-instance.xml"/>
      </xforms:instance>
    </xforms:model>
  </xhtml:head>
  ...
</xhtml:html>
```

With the example above, the resource `main-xforms-instance.xml` can contain:

```
<instance>
  <age>35</age>
  <secret>42</secret>
  <textarea>The world is but a canvas for the imagination.</textarea>
  <label>Hello, World!</label>
  <date>2004-01-07</date>
  ...
</instance>
```

The processor is configured as follows:

```
<p:processor name="oxf:xi:include" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config" href="view.xhtml"/>
  <p:output name="data" id="result"/>
</p:processor>
```

The result of the inclusion on the data output is:

```
<xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <xhtml:head>
    <xhtml:title>XForms Text Controls</xhtml:title>
    <xforms:model id="main-model" xmlns:xforms="http://www.w3.org/2002/xforms">
      <xforms:instance id="instance">
        <instance xml:base="oxf:/examples/xforms/xforms-controls/main-xforms-instance.xml">
          <age>35</age>
          <secret>42</secret>
          <textarea>The world is but a canvas for the imagination.</textarea>
          <label>Hello, World!</label>
          <date>2004-01-07</date>
          ...
        </instance>
      </xforms:instance>
    </xforms:model>
  </xhtml:head>
  ...
</xhtml:html>
```

Notice how the `xi:include` element has been replaced with the entire content of the included file. In addition, a new `xml:base` attribute has been added, as per the XInclude specification. It specifies the base URI that must be used to resolve relative URIs in the included document.

An included document may in turn use further `xi:include` elements to perform nested inclusions.

Note

The XInclude processor only supports `xi:include` with the `href` attribute and an optional `parse` attribute set to the constant `xml`. Other features of XInclude such as the `xpointer`, `encoding`, `accept` or `accept-language` attributes, and `xi:fallback` are not supported. It is planned to enhance the XInclude processor over time to support those features.

User-defined inputs allow the XInclude processor to include dynamically generated XML documents. For example, connecting the XInclude processor as follows:

```
<p:processor name="oxf:xi:include" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config" href="view.xhtml"/>
  <p:input name="my-dynamic-content" href="#some-dynamic-content"/>
  <p:output name="data" id="result"/>
</p:processor>
```

It is possible to include the XML document available on the custom `my-dynamic-content` input by using `input:*` URIs, for example with the following document as the `config` input:

```
<xhtml:html>
  <xhtml:head>
    <xhtml:title>XForms Text Controls</xhtml:title>
    <xforms:model id="main-model">
```

```

        <xforms:instance id="instance">
          <!-- This is where the inclusion is performed -->
          <xi:include href="input:my-dynamic-content"/>
        </xforms:instance>
      </xforms:model>
    </xhtml:head>
    ...
  </xhtml:html>

```

This makes the XInclude processor very versatile as a simple template processor.

4.9. XUpdate Processor

4.9.1. Scope

This section provides an overview of the XUpdate language, the OPS XUpdate engine, and the extensions to the XUpdate language supported by the OPS XUpdate engine. It also features examples programs written in XUpdate.

For more information on how to use the OPS XUpdate engine directly from Java programs, see the [XUpdate](#) section in *Integration*.

4.9.2. About XUpdate

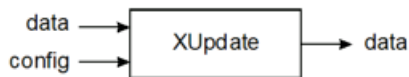
The [XML:DB group](#) has been working on a specification for a language to describes how to update an XML document. The latest version of the [XUpdate specification](#) was released in 2000 as a working draft. Since then, XUpdate has been used in a number of situations to update XML documents, in particular in XML databases like [eXist](#), [Xindice](#) and [X-Hive/DB](#).

XUpdate does not provide means to declare variables, iterate over a node set, or declare functions. In addition to the XUpdate specification, the OPS XUpdate engine implements a set of extensions to allow all of the above.

The XUpdate language was designed to be consistent with XSLT, so it would be natural for developers who are familiar with XSLT to use XUpdate. The extensions to the XUpdate specification implemented in the OPS XUpdate engine follow the same design principle as they closely mimic their XSLT counterpart.

An XUpdate transformation is similar to an XSLT transformation:

- As shown in the illustration below, just like XSLT, XUpdate transforms an XML input document into an XML output document based on a "configuration". With XSLT, the configuration is an XSLT stylesheet. With XUpdate, the configuration is a program written in the XUpdate language.



- The XSLT and XUpdate languages have similar expressive power, they both use XPath as the expression language and share elements names in a consistent way.

Every program written in XSLT could also be written in XUpdate and vice versa. A natural question would be: when is it more appropriate to use XUpdate and when is it more appropriate to use XSLT? XSLT and XUpdate are set apart by their distinctive processing model: in XSLT nodes from the input document trigger the execution of templates that define the output document, while an XUpdate provides the sequence of operations to be performed on the input document to create the output document.

As a rule of thumb, XUpdate is more appropriate when the output document is similar to the input document, while XSLT is more appropriate when the output document is a new document in which values from the input document are inserted. This can be summarized in the table below:

	XSLT	XUpdate
Execution model	Nodes from the input document trigger the execution of templates that define the output document	XUpdate provides a sequence of operations to be performed on the input document to create the output document
Most appropriate when	When the output document is a new document in which values from the input document are inserted	When the output document is similar to the input document
Typical applications	XSLT can be used as a template language, for instance to create an HTML page with both dynamic and static data. The input document contains the dynamic data. The stylesheets contains the static data and describes how the dynamic data is inserted in the document. Doing the same thing in XUpdate would be unnecessarily complicated.	XUpdate can be used as an annotation language, for instance to validate elements of an input document and to add an attribute on those elements stating if they are valid or not. Another usage is when annotating an XML document to add calculated values based on existing values. Some of the examples below do exactly that. Some cases, for instance when a parent elements has to be updated based on updates previously done to child elements, cannot be handled with XSLT 1.0 without using XSLT extensions or pipelining multiple stylesheets.

4.9.3. Language Reference

All the XUpdate elements mentioned here must be in the `http://www.xml.db.org/xupdate` namespace and for brevity we use the `xu` prefix.

The elements below are defined in the [XUpdate specification](#).

- `<xu:modifications version="1.0">`
- `<xu:value-of select="expression">`
- `<xu:if test="expression">`
- `<xu:insert-before select="expression">`
- `<xu:insert-after select="expression">`
- `<xu:append select="expression" child="expression">`
- `<xu:remove select="expression">`
- `<xu:update select="expression">`
- `<xu:variable name="qname" select="expression">`
- `<xu:element name="qname">`
- `<xu:attribute name="qname">`
- `<xu:function name="qname">` — Defines a function with the given name. The element can contain `<xu:param name="qname" select="expression">` elements that define the parameters of the function. Functions defined with `<xu:function>` are called from XPath expressions, just like regular XPath functions.

Functions in XUpdate, just like variables, can be declared anywhere, not only at the top level. In particular, they can also be declared inside other functions. The visibility rules for functions are the same as those for variables: a function has a visibility on the context where it is declared and is visible only inside the block where it is declared.

Functions in XUpdate are first-class citizen: just like variables they can be passed as arguments and returned by other functions. [An example](#) later in this section illustrates how this works.

- `<xu:choose>` — Contains one or more `<xu:when test="expression">` elements and an optional `<xu:otherwise>`. The content of the first `<xu:when>` where the test condition is true is executed. If none is true, the content of `<xu:otherwise>` is executed if present.
- `<xu:copy-of select="expression">` — Performs a copy of what is returned by the `select` expression. Unlike `<xu:value-of>`, the result of the expression is not converted to a string before it is returned.
- `<xu:for-each select="expression">` — Iterates over the node set returned by the `select` expression and executes the content of the `<xu:for-each>` for every node in the node set with that node as the current context.

Note that in the case below, the content of the `data` variable is updated, not the input document (so in this case, the input document will be left unchanged):

```
<xu:modifications xmlns:xu="http://www.xml.db.org/xupdate">
  <xu:variable name="company" select="document('oxf:/xupdate/input.xml')"/>
  <xu:for-each select="$company/company/year">
    <xu:variable name="position" select="position()"/>
    <xu:update select="/company/year[$position]">
      <xu:value-of select="@id"/>
    </xu:update>
  </xu:for-each>
  <!-- Statements using $company -->
</xu:modifications>
```

- `<xu:while select="expression">` — Executes the content of the `<xu:while>` while the condition in the `select` expression is true.
- `<xu:assign select="expression">` — Replaces the content of a variable. Once the value of a variable is set, it is possible to update parts of its contents with statements like `<xu:update>`. However no other statements we'll let you replace the root element of the variable, or the value of the variable if it is a simple type. To do so you need to use `<xu:assign>`. The new value can be either specified as an XPath expression with the `select` attribute, or in the contents of the `<xu:update>` element.
- `<xu:message>` — Just like `<xsl:message>`, logs the content of the element. For instance, to log the document being currently updated, use `<xu:message><xu:copy-of select="/" /></xu:message>`.
- `<xu:error>` — The XUpdate program is interrupted as an exception is thrown with the result of the evaluation of the element content.
- `<xu:namespace>` — Adds a namespace declaration on the current element, just like the XSLT `<xsl:namespace>`.
- The `<xu:value-of>` as described in the XUpdate specification works like the XSLT `<xsl:copy-of>` but the XUpdate specification provides no equivalent to the XSLT `<xsl:value-of>`.

We find this situation both confusing and restrictive. It is confusing because more people know about XSLT than XUpdate and they will be surprised if `<xu:value-of>` has a different behavior than the one they expected. It is restrictive because the XUpdate specification provides no equivalent to the XSLT `<xsl:value-of>` functionality.

For these reasons the OPS XUpdate engine implements both `<xu:value-of>` and `<xu:copy-of>` as in XSLT.

In a number of places you can use XPath expressions. In addition to standard XPath 1.0, you can use the following list of extension functions:

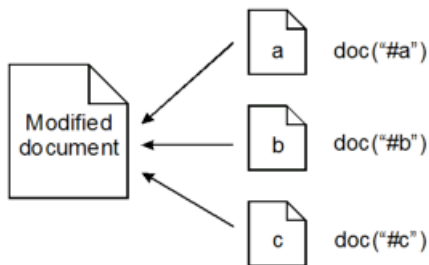
- The XPath 2.0 function `get-namespace-uri-for-prefix()` is supported (the XPath engine is still 1.0 compliant, not 2.0).
- `evaluate(xpath, namespaces, nodeset)` – Evaluates the first string argument as an XPath expression in the context of the third nodeset argument. Any prefix used in the XPath expression must be defined in the second argument, which is a nodeset of namespace nodes.

4.9.4. Using the XUpdate Processor

The XUpdate processor has the same interface as the XSLT processor. It takes two inputs: `config`, the XUpdate program, and `data`, the document to update. In XPL, the XUpdate processor can be invoked with:

```
<p:processor name="oxf:xupdate" xml:ns:p="http://www.orbeon.com/oxf/pipe/line">
  <p:input name="config" href="xupdate.xml"/>
  <p:input name="data" href="#data"/>
  <p:output name="data" id="output"/>
</p:processor>
```

It is not uncommon that a given document needs to be updated based on information stored in other documents. For example, in the situation illustrated below the document on the left is modified based on three other documents *a*, *b* and *c*.



When calling the XUpdate processor from XPL, this is done by connecting the three documents to the inputs of the XUpdate processor named *a*, *b* and *c*. Then the XUpdate program has access to those documents through XPath expressions with the `document()` or `doc()` functions using the URI *#a*, *#b* and *#c*. Note that any name can be chosen except `data` and `config` since those are reserved for the document to update and the XUpdate program. Invoking the XUpdate processor in this scenario could be done as follows:

```
<p:processor name="oxf:xupdate" xml:ns:p="http://www.orbeon.com/oxf/pipe/line">
  <p:input name="config" href="xupdate.xml"/>
  <p:input name="data" href="#data"/>
  <p:input name="a" href="x.xml"/>
  <p:input name="b" href="y.xml"/>
  <p:input name="c" href="z.xml"/>
  <p:output name="data" id="output"/>
</p:processor>
```

4.9.5. Examples

All the examples have been designed to the work on the sample document given below. This document provides numbers reflecting the sales of a company divided by quarter and by year.

```
<company>
  <year id="2000">
    <quarter id="1" sales="80"/>
    <quarter id="2" sales="56"/>
    <quarter id="3" sales="97"/>
    <quarter id="4" sales="150"/>
  </year>
  <year id="2001">
    <quarter id="1" sales="20"/>
    <quarter id="2" sales="54"/>
    <quarter id="3" sales="80"/>
    <quarter id="4" sales="90"/>
  </year>
  <year id="2002">
    <quarter id="1" sales="54"/>
    <quarter id="2" sales="65"/>
    <quarter id="3" sales="96"/>
    <quarter id="4" sales="164"/>
  </year>
</company>
```

The goal is to add a `change` attribute to the `quarter` element with the difference between the numbers for that quarter and the same quarter one year before. We also want to add a `change` attribute to the `year` element computed as the sum of the `change` attributes on its children elements. Obviously no `change` attribute can be added to the `year` and `quarter` elements of the first year. The output document looks like:

```

<company>
  <year id="2000">
    <quarter id="1" sales="80"/>
    <quarter id="2" sales="56"/>
    <quarter id="3" sales="97"/>
    <quarter id="4" sales="150"/>
  </year>
  <year id="2001" change="-139">
    <quarter id="1" sales="20" change="-60"/>
    <quarter id="2" sales="54" change="-2"/>
    <quarter id="3" sales="80" change="-17"/>
    <quarter id="4" sales="90" change="-60"/>
  </year>
  <year id="2002" change="135">
    <quarter id="1" sales="54" change="34"/>
    <quarter id="2" sales="65" change="11"/>
    <quarter id="3" sales="96" change="16"/>
    <quarter id="4" sales="164" change="74"/>
  </year>
</company>

```

What is particular in this example is that the most convenient way to implement the requirement is to first add an attribute to a set of elements (quarters) and then to update the parent elements (years) based on the values previously computed. This is typically hard to do in XSLT, but is quite natural in XUpdate:

```

<xu:modifications xmlns:xu="http://www.xml db.org/xupdate">
  <!-- Add the change attribute to the quarter elements -->
  <xu:for-each select="/company/year[position() > 1]">
    <xu:variable name="last-year" select="preceding::year"/>
    <xu:for-each select="quarter">
      <xu:variable name="quarter" select="@id"/>
      <xu:append select=" ">
        <xu:attribute name="change">
          <xu:value-of select="@sales - $last-year/quarter[@id = $quarter]/@sales"/>
        </xu:attribute>
      </xu:append>
    </xu:for-each>
  </xu:for-each>
  <!-- Add the change attribute at the year level by computing -->
  <!-- the sum of the changes at the quarter level -->
  <xu:for-each select="/company/year[position() > 1]">
    <xu:append select=" ">
      <xu:attribute name="change">
        <xu:value-of select="sum(quarter/@change)"/>
      </xu:attribute>
    </xu:append>
  </xu:for-each>
</xu:modifications>

```

The idea is to add a change attribute to the quarter elements, like in the previous example. In addition, a variation attribute must be added to the quarter elements with the difference between the change value of the current quarter and the change value of the previous quarter. Obviously, the variation attribute cannot be added to the first quarter of the second year. The output document looks like:

```

<company>
  <year id="2000">
    <quarter id="1" sales="80"/>
    <quarter id="2" sales="56"/>
    <quarter id="3" sales="97"/>
    <quarter id="4" sales="150"/>
  </year>
  <year id="2001">
    <quarter id="1" sales="20" change="-60"/>
    <quarter id="2" sales="54" change="-2" variation="58"/>
    <quarter id="3" sales="80" change="-17" variation="-15"/>
    <quarter id="4" sales="90" change="-60" variation="-43"/>
  </year>
  <year id="2002">
    <quarter id="1" sales="54" change="34" variation="94"/>
    <quarter id="2" sales="65" change="11" variation="-23"/>
    <quarter id="3" sales="96" change="16" variation="5"/>
    <quarter id="4" sales="164" change="74" variation="58"/>
  </year>
</company>

```

What is special in this example, is that the most natural way to implement the requirement is in a two-pass algorithm: first add the change attribute, then the variation attribute based on the value previously computed. Iterative algorithms can be implemented in XSLT but the exercise adds unnecessary complexity and often requires the use of XSLT extensions or multiple pipelined XSLT stylesheets. Instead, iterative algorithms can be expressed very naturally with XUpdate:

```

<xu:modifications xmlns:xu="http://www.xml db.org/xupdate">
  <xu:for-each select="/company/year[position() > 1]">
    <xu:variable name="last-year" select="preceding::year"/>
    <xu:for-each select="quarter">

```

```

<xu:variable name="quarter" select="@id"/>
<xu:append select=" " >
  <xu:attribute name="change">
    <xu:value-of select="@sales - $last-year/quarter[@id = $quarter]/@sales"/>
  </xu:attribute>
</xu:append>
</xu:for-each>
</xu:for-each>
<xu:for-each select="/company/year/quarter">
  <xu:if test="preceding::quarter/@change">
    <xu:append select=" " >
      <xu:attribute name="variation">
        <xu:value-of select="@change - preceding::quarter/@change"/>
      </xu:attribute>
    </xu:append>
  </xu:if>
</xu:for-each>
</xu:modifications>

```

The XUpdate program below adds a `change` attribute on the `quarter` elements just like the previous two examples. But, in this case, the logic to add the `change` attribute to a given element is in a function and that function is called from some code that iterates over the quarters. Here, functions are used in XUpdate just like they would be used in XSLT 2.0.

```

<xu:modifications xmlns:xu="http://www.xml-db.org/xupdate">
  <xu:function name="add-change">
    <xu:param name="last-quarter"/>
    <xu:param name="this-quarter"/>
    <xu:append select="$this-quarter">
      <xu:attribute name="change">
        <xu:value-of select="$this-quarter/@sales - $last-quarter/@sales"/>
      </xu:attribute>
    </xu:append>
  </xu:function>
  <xu:for-each select="/company/year[position() > 1]/quarter">
    <xu:variable name="id" select="@id"/>
    <xu:value-of select="add-change(preceding::year/quarter[@id = $id], .)"/>
  </xu:for-each>
</xu:modifications>

```

The OPS XUpdate engine allows for top-level functions called from XPath expressions, like XSLT 2. Additionally:

- Functions can be declared anywhere in the program, including inside other functions. The visibility rules for functions are the same as those for variables.
- Functions can be assigned to variables, passed as arguments to other functions and returned by functions. Languages providing this capability are said to treat functions as first-class citizen.

The XUpdate program below adds a `change` attribute to the `quarter` elements just like the previous examples. The `add-change` function is taken from the previous example. In addition, the logic to iterate over the quarters has been coded in a function `apply-on-quarter` that takes one argument: the function to apply to each quarter.

```

<xu:modifications xmlns:xu="http://www.xml-db.org/xupdate">
  <xu:function name="add-change">
    <xu:param name="last-quarter"/>
    <xu:param name="this-quarter"/>
    <xu:append select="$this-quarter">
      <xu:attribute name="change">
        <xu:value-of select="$this-quarter/@sales - $last-quarter/@sales"/>
      </xu:attribute>
    </xu:append>
  </xu:function>
  <xu:function name="apply-on-quarter">
    <xu:param name="f"/>
    <xu:for-each select="/company/year[position() > 1]/quarter">
      <xu:variable name="id" select="@id"/>
      <xu:value-of select="f(preceding::year/quarter[@id = $id], .)"/>
    </xu:for-each>
  </xu:function>
  <xu:value-of select="apply-on-quarter($add-change)"/>
</xu:modifications>

```

The example below uses the full power of the OPS XUpdate engine first-class citizen functions:

```

<xu:modifications xmlns:xu="http://www.xml-db.org/xupdate">
  <xu:function name="double">
    <xu:param name="f"/>
    <xu:function name="result">
      <xu:param name="x"/>
      <xu:value-of select="f($x) * 2"/>
    </xu:function>
    <xu:copy-of select="$result"/>
  </xu:function>

```

```

</xu: function>
<xu: function name="increment">
  <xu: param name="x"/>
  <xu: value-of select="$x + 1"/>
</xu: function>
<xu: variable name="incrementAndDouble" select="double($increment)"/>
<xu: update select="/">
  <result>
    <xu: value-of select="incrementAndDouble(2)"/>
  </result>
</xu: update>
</xu: modifications>

```

At the top level, the above XUpdate program defines two functions:

- `increment` simply takes a number x as argument and returns $x + 1$
- `double` is more interesting: its argument is not a number but a function f with one argument. `double` returns a function g such as $g(x) = 2 * f(x)$. Note that to return a function from another function, you must use `<xu:copy-of>` and not `<xu:value-of>`.

Finally we create a function `incrementAndDouble` by applying `double` on `increment`. As expected the result of `incrementAndDouble(2)` is 6:

```
<result>6</result>
```

4.10. SQL Processor

4.10.1. Introduction

The SQL processor provides an XML interface to any SQL database accessible through JDBC. It allows to easily query databases and produce XML outputs readily usable by other processors. Conversely, the SQL processor allows to perform updates, insertions and deletions from XML data generated by other processors.

4.10.2. Inputs and Outputs

Type	Name	Purpose	Mandatory
Input	config	Configuration template	Yes
Input	data	Source XML data	No
Input	datasource	Datasource configuration	No
Output	data	Result XML data	No

If the `data` input is not connected, the SQL processor configuration must not use XPath expressions operating on it. If this condition is not satisfied, the SQL processor generates an error at runtime. It is typically useful to omit the `data` input when the SQL processor only reads data from the database and generates an output XML document.

If the `data` output is not connected, the output of the SQL processor is simply ignored. It is typically useful to omit the `data` output when the SQL processor only updates or insert data into the database from an input XML document.

4.10.3. Configuration Template

The configuration template features a simple set of XML tags that allow you to integrate SQL queries and XML. The tags live in the `http://orbeon.org/oxf/xml/sql` namespace, which is usually mapped to the `sql` prefix. An configuration input has the following format:

```

<sql:config xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <!-- Optional user content -->
  <sql:connection>
    <!-- Datasource name (can also be specified with optional "datasource" input) -->
    <sql:datasource>my-datasource</sql:datasource>
    <!-- ... -->
  </sql:connection>
  <!-- Optional user content -->
</sql:config>

```

The `sql:datasource` element specifies a datasource name under `java:comp/env/jdbc`. In the example above, the datasource named `my-datasource` is used. How the datasource is configured depends on the application server used. Please refer to the documentation of your application server for more information.

Alternatively, the `sql:datasource` element can be omitted. In that case, the `datasource` input of the SQL processor must be connected to an external datasource definition, which describes database connections without using JNDI names mapped by the container. This is an example of datasource definition:

```

<datasource>
  <!-- Specify the driver for the database -->
  <driver-class-name>org.hsqldb.jdbcDriver</driver-class-name>
  <!-- This causes the use of the embedded database -->
  <uri>jdbc:hsqldb:file:orbeondb</uri>
  <!-- Optional username and password -->
  <username>sa</username>
  <password/>

```

```
</datasource>
```

Warning

External datasource definitions do not use connection pooling at the moment. Because creating database connections is usually an expensive operation, they should be used only for development or demo purposes.

The `sql:config` element can contain any number of user-defined content before and after the `sql:connection` element.

It is important to make sure that one and exactly one root element is output by the SQL processor. A good place to put such a root element is around the `sql:connection` element:

```
<sql:config xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <employees>
    <sql:connection>
      <!-- ... -->
    </sql:connection>
  </employees>
</sql:config>
```

The `sql:execute` element controls the execution of a single SQL query, call or update (SQL update, insert or delete). It must start with either a `sql:query`, a `sql:call` or a `sql:update` element, which contains the SQL to execute. `sql:query` and `sql:call` can be followed by zero or more `sql:result-set` elements and an optional `sql:no-results` element. Any number of `sql:execute` elements can be used under a `sql:connection` element, in order to execute several queries or updates within a single connection declaration.

```
<!-- Optional user content -->
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:query>
    <!-- ... -->
  </sql:query>
  <!-- Optional user content -->
  <sql:result-set>
    <!-- ... -->
  </sql:result-set>
  <sql:no-results>
    <!-- ... -->
  </sql:no-results>
</sql:execute>
<!-- Optional user content -->
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:call>
    <!-- ... -->
  </sql:call>
  <!-- Optional user content -->
  <sql:result-set>
    <!-- ... -->
  </sql:result-set>
  <sql:no-results>
    <!-- ... -->
  </sql:no-results>
</sql:execute>
<!-- Optional user content -->
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:update>
    <!-- ... -->
  </sql:update>
</sql:execute>
<!-- Optional user content -->
```

`sql:query` and `sql:update` encapsulate SQL statements. Like with JDBC, a distinction is made between queries (SQL `select`) and updates (SQL update, insert and delete):

```
<sql:query xmlns:sql="http://orbeon.org/oxf/xml/sql">select * from employee</sql:query>
```

```
<sql:update xmlns:sql="http://orbeon.org/oxf/xml/sql">insert into employees values ('John', 'Doe')</sql:update>
```

It is possible to pass parameters to a query using the `sql:param` element. `sql:param` requires a `type` attribute that specifies the type of the parameter to set. The type system is borrowed from the XML Schema Part 2: Datatypes specification. The following types are supported:

- `xs:string`
- `xs:int`
- `xs:boolean`
- `xs:decimal`
- `xs:float`
- `xs:double`

- xs:dateTime
- xs:date
- xs:base64Binary
- oxf:xmlFragment

Note

By default, the `xs` and `oxf` prefixes must be declared, as is customary in XML. For backward compatibility with versions of OPS where the prefixes did not have to be declared, set the `legacy-implicit-prefixes` property as follows:

```
<property as="xs:boolean" processor-name="oxf:sql" name="legacy-implicit-prefixes" value="true"/>
```

When this property is missing or set to `false`, type prefixes must be mapped as is customary for XML vocabularies. Add the following namespace declarations: `xmlns:xs="http://www.w3.org/2001/XMLSchema"` and `xmlns:odt="http://orbeon.org/oxf/xml/datatypes"`. Doing so then allows using data types as before, for example `xs:string` or `odt:xmlFragment`.

`xs:date` and `xs:dateTime` should be in one of the following formats:

- CCYY-MM-DDThh:mm:ss.sss
- CCYY-MM-DDThh:mm:ss
- CCYY-MM-DD

Unless a getter is nested in the `sql:param` element (see the section about nested queries below), a `select` attribute is mandatory. Its content is evaluated as an XPath expression against the input XML document:

```
<sql:query xmlns:sql="http://orbeon.org/oxf/xml/sql">
  select first_name, last_name from employee where employee_id in (<sql:param type="xs:int" select="/query/employee-id[1]"/>, <sql:param type="xs:int" select="/query/employee-id[2]"/>)
</sql:query>
```

Any number of `sql:param` elements may be used.

`sql:param` supports an optional boolean `replace` attribute, that can take the value `true` or `false` (the default). When `replace` is set to `true`, the parameter is simply replaced in the query, instead of being set on a JDBC `PreparedStatement`. This however works only with the `xs:int` and `oxf:literalString` types. This attribute is useful to dynamically generate parts of SQL queries, or to set parameters that do not allow being set via JDBC's `setYyy` methods. For example, with SQL Server:

```
<sql:query xmlns:sql="http://orbeon.org/oxf/xml/sql">
  select top <sql:param type="xs:int" select="/query/max-rows" replace="true"/> * from employee
</sql:query>
```

`sql:param` supports an optional `separator` attribute. When that attribute is present, the result of the XPath expression in the `select` attribute is interpreted as a node-set. One query parameter is set for each element in the node set, separated by the characters in the separator. For example:

```
<sql:query xmlns:sql="http://orbeon.org/oxf/xml/sql">
  select * from book where book_id in (<sql:param type="xs:int" select="/query/book-id" separator="," />)
</sql:query>
```

Assuming the input document contains:

```
<query>
  <book-id>5</book-id>
  <book-id>7</book-id>
  <book-id>11</book-id>
  <book-id>13</book-id>
</query>
```

The following equivalent query will be executed:

```
<sql:query xmlns:sql="http://orbeon.org/oxf/xml/sql">
  select * from book where book_id in (5, 7, 11, 13)
</sql:query>
```

The `sql:call` element allows for calling stored procedures using the JDBC escape syntax. `sql:call` uses the JDBC `CallableStatement` interface. The following example calls a procedure called `SalesByCategory` with two parameters:

```
<sql:call xmlns:sql="http://orbeon.org/oxf/xml/sql">
  { call SalesByCategory(<sql:param type="xs:string" select="/*/category"/>, <sql:param type="xs:int" select="/*/year"/>) }
</sql:call>
```

The brackets ("{" and "}") and the keyword `call` are part of the JDBC escape syntax which allows calling stored procedures without using a proprietary syntax.

Note

OUT and INOUT parameters are not yet supported.

These elements must be used only in conjunction with `sql:query` or `sql:call`.

A query or call may return multiple result-sets. It is possible to handle the result-sets returned by a query or call individually for each result-set, or globally for all result-sets, using the `sql:result-set` element. The optional `result-sets` attribute specifies how many result-sets are handled by a given `sql:result-set` element. If not specified, the default is one result-set. If the value is unbounded, the `sql:result-set` element handles all the remaining result-sets returned by the statement execution. Otherwise, a positive number of result-sets must be specified. For example:

```
<!-- Handle the first two result-sets -->
<sql:result-set result-sets="2" xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <my-first-result-sets>
    <sql:row-iterator>
      <row>
        <sql:get-columns format="xml"/>
      </row>
    </sql:row-iterator>
  </my-first-result-sets>
</sql:result-set>
<!-- Handle All the remaining result-sets -->
<sql:result-set result-sets="unbounded" xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <my-other-result-sets>
    <sql:row-iterator>
      <row>
        <sql:get-columns format="xml"/>
      </row>
    </sql:row-iterator>
  </my-other-result-sets>
</sql:result-set>
<!-- This will be executed if no row was returned by any result-set -->
<sql:no-result-ts xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <there-are-no-result-ts/>
</sql:no-result-ts>
```

Note

The body of the `sql:result-set` element is not executed if the result-set does not contain at least one row.

If a `sql:no-results` element is present after the `sql:result-sets` elements, its template content executes only if none of the previous `sql:result-set` elements has returned rows. `sql:no-results` may contain user-defined elements and nested queries (see below).

`sql:result-set` may contain a `sql:row-iterator` element, which is evaluated once for every row of the result set. `sql:row-iterator` typically contains user-defined content, as well as column getters such as `sql:get-column-value` and `sql:get-columns`.

```
<sql:result-set xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:row-iterator>
    <employee>
      <first-name>
        <sql:get-column-value type="xs:string" column="first_name"/>
      </first-name>
      <last-name>
        <sql:get-column-value type="xs:string" column="last_name"/>
      </last-name>
    </employee>
  </sql:row-iterator>
</sql:result-set>
```

Assuming the result set contains two rows with "John Doe" and "Billy Smith", the above code produces the following XML fragment:

```
<employee>
  <first-name>John</first-name>
  <last-name>Doe</last-name>
</employee>
<employee>
  <first-name>Billy</first-name>
  <last-name>Smith</last-name>
</employee>
```

`sql:get-column-value` takes a mandatory `column-name` attribute and an optional `type` attribute. If specified, the attribute must be compatible with the SQL type of the column being read. The type system is borrowed from the [XML Schema Part 2: Datatypes](#) specification. The following types are supported:

- `xs:string`
- `xs:int`
- `xs:boolean`

- `xs:decimal`
- `xs:float`
- `xs:double`
- `xs:dateTime`
- `xs:date`
- `xs:base64Binary`
- `oxf:xmlFragment`

Note

The same remark applies to types used on `sql:param`; the prefixes must be declared unless the `legacy-implicit-prefixes` specifies otherwise.

`xs:dateTime` returns a date in the following format: `CCYY-MM-DDThh:mm:ss.sss`.

`xs:date` returns a date in the following format: `CCYY-MM-DD`.

`oxf:xmlFragment` is a special type that gets the column as a string, parses it as an XML fragment, and embeds the resulting XML in the SQL processor output.

Note

For compatibility with XPath 1.0, `xs:float` and `xs:double` do not return values in the exponential notation. For example, instead of `1.2E10`, `12000000000` is returned.

When the number of columns returned is large, it is convenient to use `sql:get-columns`, which automatically determines what columns are available and generates elements accordingly. `sql:get-columns` takes an optional `prefix` attribute specifying the output namespace prefix to use for all the elements, and an optional `format` attribute specifying how column names are converted. It also supports any number of embedded `exclude` elements that specify columns to exclude from the result.

`sql:get-columns` supports an `all-elements` attribute. If set to `true`, an element is output for a column even if that column returns a null value. If missing or set to `false`, no element is output for a null column.

The namespace prefix, if specified, must have been mapped to a namespace URI.

If no `format` is specified, the original column names are used. Specifying the `xml` format converts all column names to lower case and transforms `"_"` into `" "`.

This example:

```
<sql:result-set xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:row-iterator>
    <employee>
      <sql:get-columns format="xml"/>
    </employee>
  </sql:row-iterator>
</sql:result-set>
```

Produces these results:

```
<employee>
  <first-name>John</first-name>
  <last-name>Doe</last-name>
</employee>
<employee>
  <first-name>Billy</first-name>
  <last-name>Smith</last-name>
</employee>
```

Not specifying the `xml` format generates the following results:

```
<employee>
  <first_name>John</first_name>
  <last_name>Doe</last_name>
</employee>
<employee>
  <first_name>Billy</first_name>
  <last_name>Smith</last_name>
</employee>
```

It is possible to exclude the `first_name` column as follows:

```
<sql:result-set xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:row-iterator>
    <employee>
      <sql:get-columns format="xml">
        <sql:exclude>first_name</sql:exclude>
      </sql:get-columns>
    </employee>
  </sql:row-iterator>
</sql:result-set>
```

```

    </empl oyee>
  </sql : row-i terator>
</sql : resul t-set>

```

This generates the following results:

```

<empl oyee>
  <l ast_name>Doe</l ast_name>
</empl oyee>
<empl oyee>
  <l ast_name>Smi th</l ast_name>
</empl oyee>

```

It is possible to retrieve result-set metadata. The following elements must be used within a `sql:column-iterator` element, unless a `column-name` or `column-index` attribute is explicitly specified:

- `sql:get-column-index`: retrieves the current column index.
- `sql:get-column-name`: retrieves the current column name.
- `sql:get-column-type`: retrieves the current column type name as returned by result-set metadata.

It is possible to dynamically generate new attributes with the `sql:attribute` element:

```

<my-el ement>
  <sql : attri bute name="i ndex" xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
    <sql : get-col umn-i ndex/>
  </sql : attri bute>
  ...
</my-el ement>

```

This will result in something like:

```

<my-el ement i ndex="3">...</my-el ement>

```

You can explicitly iterate over all the columns returned by a result-set with the `sql:column-iterator` element. A column iterator can be used under the `sql:result-set` element, or under the `sql:row-iterator` element. This allows for example easily extracting column metadata:

```

<sql : resul t-set xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
  <metadata>
    <sql : col umn-i terator>
      <col umn>
        <sql : attri bute name="i ndex">
          <sql : get-col umn-i ndex/>
        </sql : attri bute>
        <sql : attri bute name="name">
          <sql : get-col umn-name/>
        </sql : attri bute>
        <sql : attri bute name="type">
          <sql : get-col umn-type/>
        </sql : attri bute>
        <i ndex>
          <sql : get-col umn-i ndex/>
        </i ndex>
        <name>
          <sql : get-col umn-name/>
        </name>
        <type>
          <sql : get-col umn-type/>
        </type>
      </col umn>
    </sql : col umn-i terator>
  </metadata>
</sql : resul t-set>

```

As explained in the section about `sql-type` above, when text data, in particular XML data, is large, it is best stored as a CLOB type or, in the case of XML, as a native database XML data type such as the Oracle `XMLType` data type.

Reading and writing XML data is supported to and from database CLOBs and, with Oracle 9, to and from `XMLType`. The `oxf:xmlFragment` type must be used. To write XML data to a CLOB, use the `oxf:xmlFragment` type as follows:

```

<sql : execute xml ns: sql ="http://orbeon.org/oxf/xml /sql ">
  <sql : update>
    insert into test_cl ob_tabl e (cl ob_col umn) val ues (<sql : param select="/" type="oxf: xml Fragment" sql -
      type="cl ob"/>)
  </sql : update>
</sql : execute>

```

The XPath expression must return one element node. The result of the XPath expression specified in the `select` attribute is converted into a new XML document having as root element the selected element node. The document is then serialized to a character stream and stored as a CLOB.

To read a document from a CLOB column, use the `oxf:xmlFragment` type as follows:

```
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:query>
    select clob_column from test_clob_table
  </sql:query>
  <sql:result-set>
    <rows>
      <sql:row-iterator>
        <row>
          <sql:get-column-value type="oxf:xmlFragment" column="clob_column"/>
        </row>
      </sql:row-iterator>
    </rows>
  </sql:result-set>
</sql:execute>
```

For each row returned, the character data stored in the CLOB column is read as text and parsed into an XML fragment. The fragment must be well-formed, otherwise an exception is thrown. The resulting fragment is then embedded into the SQL processor output.

With Oracle 9, it is also possible to write to the native Oracle `XMLType` data type:

```
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:update>
    insert into test_xml_type_table (xml_type_column) values (<sql:param select="/" type="oxf:xmlFragment" sql -
      type="xml type"/>)
  </sql:update>
</sql:execute>
```

Note

The benefit of using the Oracle `XMLType` data type is that XML is stored in a structured way in the database. This allows creating indexes on XML data, doing partial document updates, etc. This however requires creating an XML schema. For more information, please refer to the [Oracle XML DB Developer's Guide](#).

Reading from an `XMLType` column is done the same way as with a CLOB column:

```
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:query>
    select xml_type_column from test_xml_type_table
  </sql:query>
  <sql:result-set>
    <rows>
      <sql:row-iterator>
        <row>
          <sql:get-column-value type="oxf:xmlFragment" column="xml_type_column"/>
        </row>
      </sql:row-iterator>
    </rows>
  </sql:result-set>
</sql:execute>
```

Warning

Writing to CLOB columns, as well as writing and reading to and from `XMLType` columns, is currently only supported with the following application server / database combinations:

- Tomcat 4.1 and Oracle 9
- WebLogic 8.1 and Oracle 9

Please [contact Orbeon](#) to inquire about supporting additional systems.

Reading from CLOB columns on the other hand is supported with all JDBC drivers that support the CLOB API.

When setting a parameter of type `xs:string` or `oxf:xmlFragment`, it is possible to specify an additional attribute on `sql:param:sql-type`. By default, text is written using the JDBC `setString()` method. In case the data must be stored in the database as a Character Large Object (CLOB) or other database-specific types, it is necessary to tell OPS that a different API must be used. For example, to write a string into a CLOB:

```
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:update>
    insert into test_clob_table (clob_column) values (<sql:param select="/document/text" type="xs:string" sql -
      type="clob"/>)
  </sql:update>
```

```
</sql : execute>
```

The same string can be written as a regular `varchar` type as follows:

```
<sql : execute xml ns: sql ="http://orbeon.org/oxf/xml/sql">
  <sql : update>
    insert into test_table (varchar_column) values (<sql : param select="/document/text" type="xs:string" sql -
      type="varchar"/>)
  </sql : update>
</sql : execute>
```

`varchar` is actually the default, so you can simply omit the `sql-type` and write:

```
<sql : execute xml ns: sql ="http://orbeon.org/oxf/xml/sql">
  <sql : update>
    insert into test_table (varchar_column) values (<sql : param select="/document/text" type="xs:string"/>)
  </sql : update>
</sql : execute>
```

Note

The disadvantage of using database columns of type `varchar` is that those are severely limited in size, for example 4000 bytes in the case of Oracle 9. The maximum size of CLOB columns is usually much larger, for example up to 4 GB with Oracle 9. In order to store large strings or large XML documents, it is therefore necessary to use the CLOB type.

The following values are supported for `sql-type`:

- `varchar` (the default)
- `clob`
- `xmltype` (see Reading and Writing XML Documents below)

Warning

Using the `clob` and `xmltype` SQL types is currently only supported with the following application server / database combinations:

- Tomcat 4.1 and Oracle 9
- WebLogic 8.1 and Oracle 9

Please [contact Orbeon](#) to inquire about supporting additional systems.

Reading from CLOB columns on the other hand is supported with all JDBC drivers that support the CLOB API.

Reading and writing binary data is supported to and from database Binary Large Objects (BLOBs) as well as binary types (`BINARY`, `VARBINARY` and `LONGVARBINARY` SQL types). The `xs:base64Binary` type (read and write) or the `xs:anyURI` type (write only) must be used. To write to a BLOB, use the `xs:base64Binary` type as follows:

```
<sql : execute xml ns: sql ="http://orbeon.org/oxf/xml/sql">
  <sql : update>
    insert into test_blob_table (blob_column) values (<sql : param select="/*" type="xs:base64Binary"/>)
  </sql : update>
</sql : execute>
```

The result of the XPath expression specified in the `select` attribute is converted into a character string, following the XPath semantics. That string is then interpreted as Base64-encoded data, before being written to the BLOB column. For example, the following input document:

```
<root>
  /9j/4AAQSkZJRgABAQEBygHKAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwSBwYGBw4KCggLEA4R ...
  KKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooA//2Q==
</root>
```

Is converted to the following string when the expression `/*` is applied:

```
/9j/4AAQSkZJRgABAQEBygHKAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwSBwYGBw4KCggLEA4R ...
KKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooA//2Q==
```

With `xs:anyURI`, the result of the XPath expression is converted into a string and interpreted as a URL. The URL is read, and the resulting data is stored into the BLOB column. For example:

```
<sql : execute xml ns: sql ="http://orbeon.org/oxf/xml/sql">
  <sql : update>
    insert into test_blob_table (blob_column) values (<sql : param select="/my/uri" type="xs:anyURI"/>)
  </sql : update>
</sql : execute>
```

Note

XForms file uploads typically generate URLs in XForms instances if the type chosen for the uploaded file in the XForms model is `xs:anyURI`. The advantage of using `xs:anyURI` is that large resources do not have to reside entirely in memory.

To read a BLOB or BINARY column, use the `xs:base64Binary` type as follows:

```
<sql:execute xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:query>
    select blob_column from test_blob_table
  </sql:query>
  <sql:result-set>
    <rows>
      <sql:row-iterator>
        <row>
          <sql:get-column-value type="xs:base64Binary" column="blob_column"/>
        </row>
      </sql:row-iterator>
    </rows>
  </sql:result-set>
</sql:execute>
```

This will produce the following result if the document above was written to the database first:

```
<rows>
  <row>
    /9j /4AAQSkZJRgABAQEBygHKAAD/2wBDAAQDAwQDAwQEBAQFBQQFBwSHBwYGBw4KCggLEA4R ...
    KKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooAKKKKACi i i gAooooA /2Q==
  </row>
</rows>
```

Note

Base64-encoded binary documents are widely used in OPS, in particular in the following cases:

- **Request generator:** as the result of certain types of HTML form submissions (typically file uploads) or request body submission, Base64-encoded representation of the uploaded files may be stored into the request document.
- **XForms upload:** as the result of a file upload, a Base64-encoded representation of the uploaded file may be stored into the XForms instance.
- **URL generator:** can read binary documents and produce Base64-encoded output according to the standard [binary document format](#).
- **HTTP serializer:** can convert Base64-encoded input according to the [binary document format](#) into a binary stream.
- **SQL processor:** as described in this document, it is able to read and write Base64-encoded binaries.

Warning

Writing to BLOB columns is currently only supported with the following application server / database combinations:

- Tomcat 4.1 and Oracle 9
- WebLogic 8.1 and Oracle 9

Please [contact Orbeon](#) to inquire about supporting additional systems.

Reading from BLOB columns on the other hand is supported with all JDBC drivers that support the BLOB API.

The `sql:value-of` and `sql:copy-of` elements have the same semantics as their XSLT 1.0 counterparts. They work against the SQL processor's input XML document.

Those elements support functions in the `http://orbeon.org/oxf/xml/sql` namespace. The only function supported for the moment is `sql:row-position()`, which returns, in a `sql:row-iterator`, the index of the current row in the result set, starting with row number 1. For example:

```
<sql:result-set xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <sql:row-iterator>
    <employee>
      <position>
        <sql:value-of select="sql:row-position()"/>
      </position>
      <first-name>
        <sql:get-column-value type="xs:string" column="first_name"/>
      </first-name>
      <last-name>
        <sql:get-column-value type="xs:string" column="last_name"/>
      </last-name>
    </employee>
  </sql:row-iterator>
</sql:result-set>
```

```

    </empl oyee>
  </sql : row-i terator>
</sql : resul t-set>

```

This generates the following results:

```

<empl oyee>
  <posi ti on>1</posi ti on>
  <fi rst-name>John</fi rst-name>
  <l ast-name>Doe</l ast-name>
</empl oyee>
<empl oyee>
  <posi ti on>2</posi ti on>
  <fi rst-name>Bi l l y</fi rst-name>
  <l ast-name>Smi th</l ast-name>
</empl oyee>

```

The `sql:update` element supports an optional `select` attribute. It is evaluated as an XPath expression against the input XML document. The expression must return a node-set (which may be empty). The update statement is executed once for every node returned. `select` attributes on nested `sql:param` elements are evaluated using the selected node as current node. With the following input XML document:

```

<empl oyees>
  <empl oyee>
    <fi rst-name>John</fi rst-name>
    <l ast-name>Doe</l ast-name>
  </empl oyee>
  <empl oyee>
    <fi rst-name>Bi l l y</fi rst-name>
    <l ast-name>Smi th</l ast-name>
  </empl oyee>
</empl oyees>

```

The following update inserts two rows in the database:

```

<sql : update select="/empl oyees/empl oyee" xml ns:sql ="http://orbeon.org/oxf/xml /sql ">
  insert into employee (fi rst_name, l ast_name) values (<sql : param type="xs:stri ng" select="fi rst-name"/
>, <sql : param type="xs:stri ng" select="l ast-name"/>)
</sql : update>

```

Consider the following three SQL tables organized in a tree. The level2 table references the level1 table, and the level3 table references the level2 table.

level1

level1_id	value
1	a
2	b

level2

level2_id	level1_id	value
1	1	a
2	1	b
3	2	c
4	2	d

level3

level2_id	value
1	a
1	b
2	c
2	d
3	e
3	f
4	g
4	h

A flat representation of the three tables joined on their respective foreign keys yields the following rows:

level1.value	level2.value	level3.value
a	a	a
a	a	b
a	b	c
a	b	d
b	c	e
b	c	f
b	d	g
b	d	h

Often it is useful to group results in order to output certain values only once. In a table, this can look like the following:

level1.value	level2.value	level3.value
a	a	a
		b
	b	c
		d
b	c	e
		f
	d	g
		h

A generalization of this consists in generating output of the form:

```

<result>
  <group1>
    <group1-header>
      <level 1-value>a</level 1-value>
    </group1-header>
    <group1-members>
      <group2>
        <group2-header>
          <level 2-value>a</level 2-value>
        </group2-header>
        <group2-members>
          <level 3-value>a</level 3-value>
          <level 3-value>b</level 3-value>
        </group2-members>
        <group2-footer>
          <level 2-value>a</level 2-value>
        </group2-footer>
      </group2>
      <group2>
        <group2-header>
          <level 2-value>b</level 2-value>
        </group2-header>
        <group2-members>
          <level 3-value>c</level 3-value>
          <level 3-value>d</level 3-value>
        </group2-members>
        <group2-footer>
          <level 2-value>b</level 2-value>
        </group2-footer>
      </group2>
    </group1-members>
    <group1-footer>
      <level 1-value>a</level 1-value>
    </group1-footer>
  </group1>
  <group1>
    <group1-header>
      <level 1-value>b</level 1-value>
    </group1-header>
    <group1-members>
      <group2>
        <group2-header>
          <level 2-value>c</level 2-value>
        </group2-header>
        <group2-members>
          <level 3-value>e</level 3-value>
          <level 3-value>f</level 3-value>
        </group2-members>
      </group2>
    </group1-members>
  </group1>

```

```

        </group2-members>
        <group2-footer>
          <l evel 2-val ue>c</l evel 2-val ue>
        </group2-footer>
      </group2>
    <group2>
      <group2-header>
        <l evel 2-val ue>d</l evel 2-val ue>
      </group2-header>
      <group2-members>
        <l evel 3-val ue>g</l evel 3-val ue>
        <l evel 3-val ue>h</l evel 3-val ue>
      </group2-members>
      <group2-footer>
        <l evel 2-val ue>d</l evel 2-val ue>
      </group2-footer>
    </group2>
  </group1-members>
  <group1-footer>
    <l evel 1-val ue>b</l evel 1-val ue>
  </group1-footer>
</group1>
</result>

```

There are two ways of generating such results. The first way is to use nested queries. A first query returns all the rows in the level1 table. Then, for each row returned, a second query returns all rows in the level2 table referencing the current row's level1_id. Similarly, for each row returned by that second query, a new query is done to get the relevant level3 rows. The code would look as follows:

```

<sql : config xmlns:sql="http://orbeon.org/oxf/xml/sql">
  <resultset>
    <sql : connection>
      <sql : datasource>my-datasource</sql : datasource>
    <sql : execute>
      <sql : query>
        select l evel 1.val ue val ue, l evel 1.l evel 1_i d i d from l evel 1 order by l evel 1.val ue
      </sql : query>
      <sql : result-set>
        <sql : row-iterator>
          <group1>
            <group1-header>
              <l evel 1-val ue>
                <sql : get-column-value type="xs:string" column="val ue"/>
              </l evel 1-val ue>
            </group1-header>
            <group1-members>
              <sql : execute>
                <sql : query>
                  select l evel 2.val ue val ue, l evel 2.l evel 2_i d i d from l evel 2 where l evel 2.l evel 1_i d =
                    <sql : param type="xs:int"><sql : get-column-value type="xs:int" column="i d"/></sql : param>
                  order by l evel 2.val ue
                </sql : query>
                <sql : result-set>
                  <sql : row-iterator>
                    <group2>
                      <group2-header>
                        <l evel 2-val ue>
                          <sql : get-column-value type="xs:string" column="val ue"/>
                        </l evel 2-val ue>
                      </group2-header>
                      <group2-members>
                        <sql : execute>
                          <sql : query>
                            select l evel 3.val ue val ue from l evel 3 where l evel 3.l evel 2_i d =
                              <sql : param type="xs:int"><sql : get-column-value type="xs:int" column="i d"/></sql : param>
                            order by l evel 3.val ue
                          </sql : query>
                          <sql : result-set>
                            <sql : row-iterator>
                              <l evel 3-val ue>
                                <sql : get-column-value type="xs:string" column="val ue"/>
                              </l evel 3-val ue>
                            </sql : row-iterator>
                          </sql : result-set>
                        </sql : execute>
                      </group2-members>
                    </group2>
                  <group2-footer>
                    <l evel 2-val ue>
                      <sql : get-column-value type="xs:string" column="val ue"/>
                    </l evel 2-val ue>
                  </group2-footer>
                </group2>
              </sql : row-iterator>
            </group1>
          </sql : result-set>
        </sql : execute>
      </group1>
    </sql : result-set>
  </resultset>
</sql : config>

```



```

        </sql: result-set>
      </sql: execute>
    </group1-members>
  </group1-footer>
  <level 1-value>
    <sql: get-column-value type="xs:string" column="value"/>
  </level 1-value>
</group1-footer>
</group1>
</sql: row-iterator>
</sql: result-set>
</sql: execute>
</sql: connecti on>
</result s>
</sql: confi g>

```

A nested query can access parameters from the input XML document like any regular query. It can also access results from outer queries by nesting a getter in a `sql:param` element. In that case, getters can take an optional `ancestor` attribute that specifies which level of outer query to access. If omitted, the `ancestor` attribute takes the value 1 when used in a `sql:query` or `sql:update`, which means the first outer query; it defaults to the value 0 when used in a `sql:row-iterator`, which means the query at the current level.

While nested queries have their uses, in the example above 3 queries have to be written and no less than 7 queries are executed to produce the final result. It can be elegantly rewritten using the `sql:group` and `sql:member` elements. `sql:group` has to be the first element under a `sql:row-iterator` or `sql:member` element. It takes a mandatory `column-name` attribute that specifies the name of the column on which grouping is done.

For every group, a header is output only once. Then, the content under the `sql:member` element is output for each row. Finally, the footer is output. The header and the footer can access columns. There is no limit in as to how deep grouping can be done.

The code below generates with a single SQL query the same results as the example above:

```

<sql: confi g xmlns: sql ="http://orbeon.org/oxf/xml/sql ">
  <result s>
    <sql: connecti on>
      <sql: datasource>my-datasource</sql: datasource>
    <sql: execute>
      <sql: query>
        select level 1. value v1, level 2. value v2, level 3. value v3 from level 1, level 2, level 3 where
        level 1. level 1_id = level 2. level 1_id and level 2. level 2_id = level 3. level 2_id order by level 1. value,
        level 2. value, level 3. value
      </sql: query>
      <sql: result-set>
        <sql: row-iterator>
          <sql: group column="v1">
            <group1>
              <group1-header>
                <level 1-value>
                  <sql: get-column-value type="xs:string" column="v1"/>
                </level 1-value>
              </group1-header>
              <group1-members>
                <sql: member>
                  <sql: group column="v2">
                    <group2>
                      <group2-header>
                        <level 2-value>
                          <sql: get-column-value type="xs:string" column="v2"/>
                        </level 2-value>
                      </group2-header>
                      <group2-members>
                        <sql: member>
                          <level 3-value>
                            <sql: get-column-value type="xs:string" column="v3"/>
                          </level 3-value>
                        </sql: member>
                      </group2-members>
                    </group2>
                  </sql: group>
                </group1-members>
              </group1-footer>
                <level 1-value>
                  <sql: get-column-value type="xs:string" column="v1"/>
                </level 1-value>
              </group1-footer>
            </group1>
          </sql: group>
        </sql: row-iterator>
      </sql: result-set>
    </sql: execute>
  </result s>
</sql: confi g>

```

```

    </sql : connecti on>
  </resul ts>
</sql : confi g>

```

Note that correct ordering of the rows in the SQL query is important because headers and footers are output when the columns on which grouping is done change, in the order returned by the result set.

Like in XSLT 1.0, text nodes in the configuration containing only whitespace characters are stripped. Text nodes that contain at least one non-whitespace character are not stripped and copied to the output.

To better control the output of text, the `sql:text` element is provided. It is similar to the `xsl:text` element. `xsl:text` encapsulate text that is output as is. In particular, it can encapsulate all whitespace characters.

4.10.4. Transactions Management

OPS executes each HTTP request in its own transaction. If a request fails for any reason, the SQL Processor rolls back the transaction. The transaction is committed only when the pipeline execution in complete.

4.11. XML Databases

4.11.1. Introduction

XML databases allow you to easily store, index and query XML documents. They are especially useful when dealing with *document-centric* applications, where the structure cannot be mapped naturally to a relational database.

OPS integrates with Tamino XML Server databases, as well as XML:DB databases such as eXist.

4.11.2. Tamino XML Server 4.1

Software AG's [Tamino](#) provides a complete XML storage solution. OPS allows you to easily store, query, update, and delete documents in Tamino. The following sections describe the four OPS processors for Tamino.

All Tamino processors have a common `config` input, describing the database connection and collection.

The configuration of the Tamino processors can be done in two ways: either system-wide via the [OPS Properties](#), or locally for a specific instance of the processor through the `config` input. The local configuration takes precedence if available.

Note

The `collection` configuration elements cannot be specified system-wide.

The `config` input document specifies the URL of the Tamino server, the credentials to use when connecting and the collection to access. The following table describes the configuration elements.

Name	Description
url	Tamino database URL.
username	Username to authenticate with the server
password	Password to authenticate with the server
collection	XML Collection to use

This RelaxNG schema describes the expected document.

```

<el ement name="confi g" datatypeLi brary="http: //www. w3. org/2001/XMLSchema-datatypes">
  <i nterl eave>
    <opti onal >
      <el ement name="url ">
        <data type="anyURI "/>
      </el ement>
    </opti onal >
    <opti onal >
      <el ement name="username">
        <data type="stri ng"/>
      </el ement>
    </opti onal >
    <opti onal >
      <el ement name="password">
        <data type="stri ng"/>
      </el ement>
    </opti onal >
    <el ement name="coll ecti on">
      <data type="stri ng"/>
    </el ement>
  </i nterl eave>
</el ement>

```

The Tamino processors can be configured in the [OPS properties file](#), allowing all instances to share the same configuration. The following processor properties are allowed:

Name	Type	Description
url	anyURI	Tamino Server URL.
username	string	Username to authenticate with the server.
password	string	Password to authenticate with the server.

These properties are set as follows:

```
<property as="xs:anyURI" processor-name="oxf:tami no-query" name="url" value="http://localhost/tami no/welcome_4_1_4"/>
<property as="xs:string" processor-name="oxf:tami no-query" name="username" value="..."/>
<property as="xs:string" processor-name="oxf:tami no-query" name="password" value="..."/>
```

The following global properties are allowed:

Name	Type	Description
oxf.tamino.isolation-degree	string	Isolation degree. Possible values are: uncommittedDocument, committedCommand, stableCursor, stableDocument, serializable. See the Tamino documentation for more details on the isolation degree.
oxf.tamino.lock-mode	string	Lock mode. Possible values are unprotected, shared, protected. See the Tamino documentation for more details on the lock mode.

These properties as set as follows:

```
<property as="xs:string" name="oxf.tami no.i sol ati on-degree" value="..."/>
<property as="xs:string" name="oxf.tami no.l ock-mode" value="..."/>
```

The `oxf:tamino-query` processor processes queries either using Tamino's X-Query or W3C XQuery.

The data input contains only the root element and the query. The root element is either `query` for an X-Query query, or `xquery` for an XQuery query.

The processor sends the result of the query in the data output. The root element is always `result`.

```
<p:processor name="oxf:tami no-query" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://localhost/tami no/welcome_4_1_4</url>
      <username>tami no</username>
      <password>password</password>
      <collection>encyclopedia</collection>
    </config>
  </p:input>
  <p:input name="data">
    <query>
      /jazzMusician[@ID="ParkerCharlie"]
    </query>
  </p:input>
  <p:output name="data" id="result"/>
</p:processor>
```

```
<p:processor name="oxf:tami no-query" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://localhost/tami no/welcome_4_1_4</url>
      <username>tami no</username>
      <password>password</password>
      <collection>encyclopedia</collection>
    </config>
  </p:input>
  <p:input name="data">
    <xquery>
      for $m in input()/jazzMusician, $c in input()/collaboration, $a in input()/album where $m@ID= $c/
      jazzMusician and $c/result = $a/productNo return
      <musician>
        {$m/name}<album>{$a/title}</album>
      </musician>
    </xquery>
  </p:input>
  <p:output name="data" id="result"/>
```

```
</p:processor>
```

The `oxf:tamino-insert` processor allows you to insert a document in Tamino.

Note

You need to make sure that the document conforms to one of the registered schemas in the current collection. If Tamino can't validate the document, an exception is thrown.

The data input contains the document to insert.

```
<p:processor name="oxf:tamino-insert" xmlns:p="http://www.oregon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://localhost/tamino/welcome_4_1_4</url>
      <username>tamino</username>
      <password>password</password>
      <collection>encyclopedica</collection>
    </config>
  </p:input>
  <p:input name="data">
    <jazzMusician ID="DavisMiles" type="instrumentalist">
      <name>
        <first>Miles</first>
        <last>Davis</last>
      </name>
      <birthDate>1926-05-26</birthDate>
      <instrument>trumpet</instrument>
    </jazzMusician>
  </p:input>
</p:processor>
```

The `oxf:tamino-delete` processor allows you to remove documents from Tamino.

The data input contains the X-Query to select the document(s) to be removed.

```
<p:processor name="oxf:tamino-delete" xmlns:p="http://www.oregon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://localhost/tamino/welcome_4_1_4</url>
      <username>tamino</username>
      <password>password</password>
      <collection>encyclopedica</collection>
    </config>
  </p:input>
  <p:input name="data">
    <query>
      /jazzMusician[@ID="DavisMiles"]
    </query>
  </p:input>
</p:processor>
```

The `oxf:tamino-update` processor allows you to update parts of documents directly inside Tamino. An extension of XQuery is used for that purpose. Refer to the Tamino documentation for more information.

The data input contains the XQuery expression to update one or multiple nodes.

```
<p:processor name="oxf:tamino-update" xmlns:p="http://www.oregon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>http://localhost/tamino/welcome_4_1_4</url>
      <username>tamino</username>
      <password>password</password>
      <collection>encyclopedica</collection>
    </config>
  </p:input>
  <p:input name="data">
    <query>
      update replace input() /jazzMusician[@ID="ParkerCharlie"] /instrument with <instrument>piano</instrument>
    </query>
  </p:input>
</p:processor>
```

OPS initiates a transaction for every HTTP request, and commits the transaction when the pipeline is executed normally. If an exception occurs in the pipeline, the transaction is rolled back.

4.11.3. XML:DB Databases (eXist)

Note

This documentation has not yet been written.

4.12. LDAP Processor

4.12.1. Introduction

The LDAP Processor allows OPS to query an LDAP directory server. The LDAP processor uses the [Java Naming and Directory Interface \(JNDI\)](#) and should work with all compatible servers. However, it has only been tested with [Sun ONE Directory Server](#) and [Open LDAP](#).

4.12.2. Usage

You instantiate the LDAP Processor with the processor URI `oxf/processor/ldap`. The Processor takes two inputs, `config` and `filter`, and one output, `data`.

The configuration of the LDAP Processor can be done in two ways: either system-wide via the [OPS Properties](#), or locally for a specific instance of the processor through the `config` input. The local configuration takes precedence if available.

Note

The `root-dn` and `attribute` configuration elements cannot be specified system-wide.

The `config` input document specifies the host name and port number of the LDAP server, as well as the credentials to use when connecting. The following table describes the configuration elements.

Name	Description
host	LDAP Server Host
port	LDAP Server Port Number
protocol	Protocol to connect to the server, eg. SSL
bind-dn	Distinguished Name to authenticate with the server
password	Password to authenticate with the server
root-dn	Root DN to bind to.
attribute	LDAP attributes to include in the LDAP response. If no attribute is specified, all returned attributes are included.

This RelaxNG schema describes the expected document.

```
<element name="config" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <interleave>
    <optional>
      <element name="host">
        <data type="string"/>
      </element>
    </optional>
    <optional>
      <element name="port">
        <data type="integer"/>
      </element>
    </optional>
    <optional>
      <element name="bind-dn">
        <data type="string"/>
      </element>
    </optional>
    <optional>
      <element name="password">
        <data type="string"/>
      </element>
    </optional>
    <element name="root-dn">
      <data type="string"/>
    </element>
    <optional>
      <element name="protocol">
        <data type="string"/>
      </element>
    </optional>
    <zeroOrMore>
      <element name="attribute">
        <data type="string"/>
      </element>
    </zeroOrMore>
  </interleave>
</element>
```

```

    </zeroOrMore>
  </interleave>
</element>

```

The LDAP Processor can be configured through the [OPS Properties](#), allowing all instances to share the same configuration. The following properties are allowed:

Name	Description
oxf.processor.ldap.host	LDAP Server Host
oxf.processor.ldap.port	LDAP Server Port Number
oxf.processor.ldap.protocol	Protocol to connect to the server, eg. SSL
oxf.processor.ldap.bind-dn	Distinguished Name to authenticate with the server
oxf.processor.ldap.password	Password to authenticate with the server

The `filter` input takes the LDAP query sent to the server. The single `filter` element contains a query string that follows the standard LDAP filter syntax specified in [RFC 2254](#).

```

<element name="filter" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <data type="string"/>
</element>

```

The LDAP processor outputs the query results in its `data` output. The resulting document looks like the example below:

```

<results>
  <result>
    <name>cn=John Smith</name>
    <attribute>
      <name>sn</name>
      <value>Smith</value>
    </attribute>
    [...]
  </result>
  [...]
</results>

```

4.12.3. Example

The following example shows a basic LDAP query. The LDAP Processor connects to an LDAP server on the same machine using the administrator account to log in. It then queries the server for objects containing a `uid` attribute with the `12345` value. Only the `cn` and `description` attributes are returned.

```

<p:processor name="oxf:ldap" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <host>localhost</host>
      <port>389</port>
      <bind-dn>cn=Directory Manager</bind-dn>
      <password>abcdef</password>
      <root-dn>o=Company.com</root-dn>
      <attribute>cn</attribute>
      <attribute>description</attribute>
    </config>
  </p:input>
  <p:input name="filter">
    <filter>(uid=12345)</filter>
  </p:input>
  <p:output name="data" id="ldap-results"/>
</p:processor>

```

4.13. Directory Scanner

4.13.1. Introduction

The purpose of the Directory Scanner processor is to analyse a directory structure in a filesystem and to produce an XML document containing metadata about the files, such as name and size. It is possible to specify which files and directories to include and exclude in the scanning process. The Directory Scanner is also able to optionally retrieve image metadata.

4.13.2. Inputs and Outputs

Type	Name	Purpose	Mandatory
Input	config	Configuration	Yes

Output	data	Result XML data	Yes
--------	------	-----------------	-----

The Directory Scanner is typically called this way from XPL pipelines:

```
<p: processor name="oxf: directory-scanner" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <!-- The configuration can often be inline -->
  <p: input name="config">...</p: input>
  <p: output name="data" id="directory-scan"/>
</p: processor>
```

4.13.3. Configuration

The config input configuration has the following format:

```
<config>
  <base-directory>file:/</base-directory>
  <include>**/*.x?l</include>
  <include>**/*.xhtml</include>
  <include>**/*.java</include>
  <exclude>example-descriptor.xml</exclude>
  <case-sensitive>false</case-sensitive>
</config>
```

Element	Purpose	Format	Default
base-directory	Directory under which files and directories are scanned, referred to below as the search directory.	A file: or oxf: URL. The URL may be relative to the location of the containing XPL file. <div>Note The oxf: protocol works only with resource managers that allow accessing the actual path of the file. These include the Filesystem and WebApp resource manager.</div>	None.
include	Specifies which files are included	Apache Ant pattern .	None.
exclude	Specifies which files are excluded	Apache Ant pattern .	None.
case-sensitive	Whether include and exclude patterns are case-sensitive.	true or false.	true
default-excludes	Whether a set of default exclusion rules must be automatically loaded. The list is as follows: <ul style="list-style-type: none"> Miscellaneous typical temporary files <ul style="list-style-type: none"> **/*~ **/### **/.#* **/%*%* **/._* CVS <ul style="list-style-type: none"> **/CVS **/CVS/** **/.cvsignore SCCS <ul style="list-style-type: none"> **/SCCS **/SCCS/** Visual SourceSafe <ul style="list-style-type: none"> **/vssver.scc Subversion <ul style="list-style-type: none"> **/.svn **/.svn/** Mac <ul style="list-style-type: none"> **/.DS_Store 	true or false.	false
image-metadata/basic-info	Whether basic image metadata must be extracted.	true or false.	false

image-metadata/exif-info	Whether Exif image metadata must be extracted.	true or false.	false
image-metadata/iptc-info	Whether iptc image metadata must be extracted.	true or false.	false

4.13.4. Output Format

The image format starts with a root `directory` element with a `name` and `path` attribute. The `name` attribute specifies the name of the search directory, e.g. `web`. The `path` attribute specifies an absolute path to that directory.

The root element then contains a hierarchical structure of `directory` and `file` elements found. For example:

```
<directory name="address-book" path="c:\Documents and Settings\John Doe\OPS\src\examples\web\examples\address-book">
  <directory name="initialization" path="initialization">
    <file last-modified-ms="1101487772375" last-modified-date="2004-11-26T17:49:32.375" size="1250"
      path="initialization\init-database.xml" name="init-database.xml"/>
    <file last-modified-ms="1101512191718" last-modified-date="2004-11-27T00:36:31.718" size="2410"
      path="initialization\init-script.xml" name="init-script.xml"/>
  </directory>
  <file last-modified-ms="1101488200406" last-modified-date="2004-11-26T17:56:40.406" size="5618"
    path="model.xml" name="model.xml"/>
  <file last-modified-ms="1101484041437" last-modified-date="2004-11-26T16:47:21.437" size="941" path="page-flow.xml"
    name="page-flow.xml"/>
  <file last-modified-ms="1121104181591" last-modified-date="2005-07-11T19:49:41.591" size="3165"
    path="view.xml" name="view.xml"/>
  <file last-modified-ms="1093118707000" last-modified-date="2004-08-21T22:05:07.000" size="934" path="xforms-model.xml"
    name="xforms-model.xml"/>
</directory>
```

`directory` elements contain basic information about a matched directory:

Name	Value
path	Path to the directory, relative to the parent directory. Includes the current directory name.
name	Local directory name.

Note

The `path` attribute on the root element is an absolute path from a filesystem root. The `path` on child `directory` element are relative to their parent `directory` element.

`file` elements contain basic information about a matched file:

Name	Value
last-modified-ms	Timestamp of last modification in milliseconds.
last-modified-date	Timestamp of last modification in XML <code>xs:dateTime</code> format.
size	Size of the file in bytes.
path	Path to the file, relative to the parent directory. Includes the file name.
name	Local file name.

When the configuration's `image-metadata` element is specified, metadata about images is extracted.

Note

Images are identified by reading the beginning of the files. This means that extracting image metadata is usually more expensive in time than just producing regular file metadata.

When an image is identified, an `image-metadata` element is available under the corresponding `file` element:

When `image-metadata/basic-info` is `true` in the configuration, a `basic-info` element is created under `image-metadata`:

Element Name	Element Value
content-type	Media type of the file: <code>image/jpeg</code> , <code>image/gif</code> , <code>image/png</code> . Other <code>image/*</code> values may be produced for other image formats.
width	Image width, if found.

height	Image height, if found.
comment	Image comment, if found (JPEG only).

When `image-metadata/exif-info` is true in the configuration, zero or more `exif-info` elements are created under `image-metadata`. Each element has an attribute containing the name of the category of Exif information. Basic Exif information has the name `Exif`. Other names may include `Canon` `Makernote` for a Canon camera, `Interoperability`, etc. Under each `exif-info` element, zero or more `param` elements are contained, with the following sub-elements:

Element Name	Element Value
id	The Exif parameter id. For example, 271 denotes the make of the camera
name	A default English name for the given parameter id, when known, for example <code>Make</code> .
value	The value of the parameter, for example <code>Canon</code> .

This is an example of file element with image metadata:

```
<file last-modified-ms="1120343217984" last-modified-date="2005-07-03T00:26:57.984" size="961130"
path="image0001.jpg" name="image0001.jpg">
  <image-metadata>
    <basic-info>
      <content-type>image/jpeg</content-type>
      <width>2272</width>
      <height>1704</height>
    </basic-info>
    <exif-info name="Exif">
      <param>
        <id>271</id>
        <name>Make</name>
        <value>Canon</value>
      </param>
      <param>
        <id>272</id>
        <name>Model</name>
        <value>Canon PowerShot S40</value>
      </param>
      ...
    </exif-info>
    ...
  </image-metadata>
</file>
```

When `image-metadata/iptc-info` is true in the configuration, zero or more `iptc-info` elements are created under `image-metadata`. Each element has an attribute containing the name of the category of IPTC information. The children element of `iptc-info` are the same as for `exif-info`.

The Directory Scanner does not provide metadata about other files at the moment, but the processor could be extended to support more metadata, about image formats but also about other file formats such as sound files, etc.

4.13.5. Ant Patterns

Note

This section of the documentation is reproduced from a section of the [Apache Ant Manual](#), with minor adjustments.

Patterns are used for the inclusion and exclusion of files. These patterns look very much like the patterns used in DOS and UNIX:

`***` matches zero or more characters, `?` matches one character.

In general, patterns are considered relative paths, relative to a task dependent base directory (the `dir` attribute in the case of `<fileset>`). Only files found below that base directory are considered. So while a pattern like `../foo.java` is possible, it will not match anything when applied since the base directory's parent is never scanned for files.

Examples:

- `*.java` matches `.java`, `x.java` and `FooBar.java`, but not `FooBar.xml` (does not end with `.java`).
- `? . java` matches `x.java`, `A.java`, but not `.java` or `xyz.java` (both don't have one character before `.java`).

Combinations of `*`'s and `?`'s are allowed.

Matching is done per-directory. This means that first the first directory in the pattern is matched against the first directory in the path to match. Then the second directory is matched, and so on. For example, when we have the pattern `/?abc/*/*.java` and the path `/xabc/foobar/test.java`, the first `?abc` is matched with `xabc`, then `*` is matched with `foobar`, and finally `*.java` is matched with `test.java`. They all match, so the path matches the pattern.

To make things a bit more flexible, we add one extra feature, which makes it possible to match multiple directory levels. This can be used to match a complete directory tree, or a file anywhere in the directory tree. To do this, `**` must be used as the name of a directory. When `**` is used as the name of a directory in the pattern, it matches zero or more directories. For example: `/test/**` matches all files/directories under `/test/`, such as `/test/x.java`, or `/test/foo/bar/xyz.html`, but not `/xyz.xml`.

There is one "shorthand" - if a pattern ends with / or \, then ** is appended. For example, mypackage/test/ is interpreted as if it were mypackage/test/**.

Example patterns:

** /CVS/ *	<p>Matches all files in CVS directories that can be located anywhere in the directory tree.</p> <p>Matches:</p> <pre>CVS/Repository org/apache/CVS/Entries org/apache/jakarta/tools/ant/CVS/Entries</pre> <p>But not:</p> <pre>org/apache/CVS/foo/bar/Entries</pre> <p>(foo/bar/ part does not match)</p>
org/apache/jakarta/**	<p>Matches all files in the org/apache/jakarta directory tree.</p> <p>Matches:</p> <pre>org/apache/jakarta/tools/ant/docs/index.html org/apache/jakarta/test.xml</pre> <p>But not:</p> <pre>org/apache/xyz.java</pre> <p>(jakarta/ part is missing).</p>
org/apache/**/CVS/ *	<p>Matches all files in CVS directories that are located anywhere in the directory tree under org/apache.</p> <p>Matches:</p> <pre>org/apache/CVS/Entries org/apache/jakarta/tools/ant/CVS/Entries</pre> <p>But not:</p> <pre>org/apache/CVS/foo/bar/Entries</pre> <p>(foo/bar/ part does not match)</p>
** /test/**	<p>Matches all files that have a test element in their path, including test as a filename.</p>

When these patterns are used in inclusion and exclusion, you have a powerful way to select just the files you want.

4.14. Delegation Processor

4.14.1. Introduction

The Delegation processor allows calling services implemented as:

- A JavaBean
- An EJB
- A Web service

The main benefit of the Delegation processor is that you do not need to implement your own XML processor in Java to call services.

4.14.2. Inputs and Outputs

Type	Name	Purpose	Mandatory
Input	interface	<p>The <code>interface</code> input declares services that you can then call in the <code>call</code> input. You define one or more service identifiers and map those to a given JavaBean, EJB or Web service. You also need to provide information that the delegation processor needs to call the service: for instance a JNDI name for an EJB or a class name for a JavaBean.</p> <p>The <code>interface</code> input is the only document with information specific to the service type (EJB, Web service or JavaBean). This means that if, at some point, a service is moved from, say, a JavaBean to an EJB, only the interface has to be modified, but not the <code>call</code> input.</p>	Yes

Input	call	<p>The call input contains an XML document which is a template containing <delegation:execute> elements. Each <delegation:execute> element specifies a service to be called and the parameters to be sent to that service.</p> <div data-bbox="352 195 1352 342" data-label="Text"> <p>Note</p> <p>For calls to JavaBeans and EJBs in particular, it is important that the resulting document be a well-formed XML document. In particular, the resulting document must have a root element. In this case, always be sure that your call input contains at least one root element around <delegation:execute> elements.</p> </div>	Yes
Output	data	The data output produces a document based on the call input where the <delegation:execute> elements have been replaced with the value returned by the service.	Yes

4.14.3. Calling a JavaBean

This is an example of using the Delegation processor to call a JavaBean:

```
<p:processor name="oxf:delegation">
  <p:input name="interface">
    <config>
      <service id="my-service" type="javaBean" class="MyClass"/>
    </config>
  </p:input>
  <p:input name="call">
    <result>
      <delegation:execute service="my-service" operation="myMethod">
        <param1 xsi:type="xs:string">param1</param1>
      </delegation:execute>
    </result>
  </p:input>
  <p:output name="data" id="result"/>
</p:processor>
```

- The interface input declares the JavaBean to call:
 - The value of the type attribute must be set to javaBean.
 - The class attribute references the Java class to instantiate.
- The call input defines the method to call and the parameters to pass:
 - The service attribute references the service id declared in the interface input.
 - Each child element of <delegation:execute> corresponds to a parameter of the method to be called, in the order they appear in the method signature. The value in the xsi:type attribute must match the parameter type. The content of the element is the value passed to the Java method.

Note

Note: the only two supported types are xs:string and xs:double.

4.14.4. Calling an EJB

This is an example of using the Delegation processor to call an EJB:

```
<p:processor name="oxf:delegation">
  <p:input name="interface">
    <config>
      <service id="creditcard-validation" type="stateless-ejb" uri="java:comp/env/ejb/creditcard-validation"/>
    </config>
  </p:input>
  <p:input name="call">
    <delegation:execute service="creditcard-validation" operation="validate">
      <number xsi:type="xs:string">1234123412341234</number>
      <type xsi:type="xs:string">visa</type>
    </delegation:execute>
  </p:input>
  <p:output name="data" id="result"/>
</p:processor>
```

- The interface input declares the EJB that will be called:
 - The value of the type attribute must be set to stateless-ejb.
 - The uri attribute references the JNDI name of the EJB.
- The call input defines the method to be called and the parameters:
 - The service attribute references the service id declared in the interface input.

- Each child element of `<delegation:execute>` corresponds to an attribute of the method to be called. The name of each element must match the attribute name, and the value in the `xsi:type` attribute must match the attribute type. The content of the element is the value passed to the EJB method.

4.14.5. Calling a Web Service

```
<p:processor name="oxf:delegation">
  <p:input name="interface">
    <config>
      <service id="quotes" type="webservice" style="rpc" endpoint="http://www.scdi.org/~avernet/webservice/">
        <operation nsuri="urn:avernet" name="getRandomQuote" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </service>
    </config>
  </p:input>
  <p:input name="call">
    <delegation:execute service="quotes" operation="getRandomQuote" />
  </p:input>
  <p:output name="data" id="result" />
</p:processor>
```

```
<p:processor name="oxf:delegation">
  <p:input name="interface">
    <config>
      <service id="stock-quote" type="webservice" style="document" endpoint="http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx">
        <operation name="get-quote" soap-action="http://ws.cdyne.com/GetQuote" />
      </service>
    </config>
  </p:input>
  <p:input name="call">
    <delegation:execute service="stock-quote" operation="get-quote" xsi:version="2.0">
      <m:GetQuote>
        <m:StockSymbol>
          <xsi:value-of select="/symbol" />
        </m:StockSymbol>
        <m:LicenseKey>0</m:LicenseKey>
      </m:GetQuote>
    </delegation:execute>
  </p:input>
  <p:output name="data" id="result" />
</p:processor>
```

- The interface input declares the Web service to be called:
 - Valid values for the style attribute are `rpc` and `document`. The default value if the attribute is missing is `rpc`.
 - The `id` (referenced when Web service is called in the `call` input) and `endpoint` (the SOAP endpoint) attributes are mandatory.
 - Declaring the Web service operations is optional for document-style services. You only need to do so if you want to specify a SOAP action for a given operation.
 - Optionally you can declare for each operation the encoding style (`encodingStyle` attribute) and SOAP action (`soap-action` attribute).
 - Optionally you can declare what part of the SOAP result document is returned by specifying an XPath expression in the optional `select` attribute on the operation element. If you don't specify an expression, by default when the style is RPC the content of the first child element under the SOAP body is returned, and when the style is document the content of the SOAP body is returned.
- In the `call` input:
 - The XML fragment you specify in the `execute` element is sent as is, and you are responsible of making sure that it is valid according to the encoding style.
 - Referencing a specific operation in the `operation` attribute is mandatory for RPC-style services, and is optional for document-style services (you only want to do so if you declared specific information about the operation in the `interface` input).
 - The optional `timeout` attribute on the `execute` element specifies a connection timeout in milliseconds. The value must be a non-negative integer, with the value 0 meaning no timeout. If not specified, the timeout value is the default timeout of the underlying web service implementation.

4.15. Java Processor

4.15.1. Rationale

OPS comes with a number of pre-built processors. However, in some cases, it makes sense for the developer to write a new processor in Java. For the purpose of this discussion, let's assume that the processor is implemented in a file called `MyProcessor.java`. Custom processors can essentially be deployed in two ways:

- "Manually":
 - Compile `MyProcessor.java`.
 - Place the compiled class in `WEB-INF/classes` (or in any other location where it can be found by the classloader used to load the classes in `ops.jar`).
 - Declare the new processor in `processors.xml` (i.e. mapping an URI to this new processor).

4. Use the newly declared processor in an XPL (OPS Pipeline Definition Language) file using the URI declared in the `processors.xml`.

- Using the Java processor:
 1. Place `MyProcessor.java` with the other resources.
 2. Use the new processor in an XPL file through the Java processor. (We'll see below how this processor works in detail.)

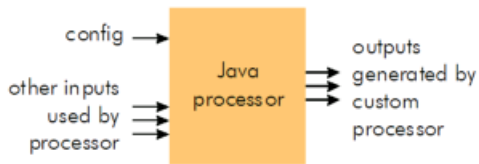
4.15.2. Benefits and Drawbacks

The main advantages of using the Java processor versus manually compiling and deploying the compiled processor are:

- Easy deployment: the Java file is placed with the other resources and one does not need to worry about compilation, packaging and deployment.
- Immediate visibility upon modifications: one can change the Java file, save it and instantly see the result in the browser (no need to compile, redeploy the application).

However, one should also note the drawbacks that come with the Java processor. In particular, the XML code needed in the XPL file to call a custom processor using the Java processor is a bit more complex than the XML code used to call a processor declared in `processors.xml`.

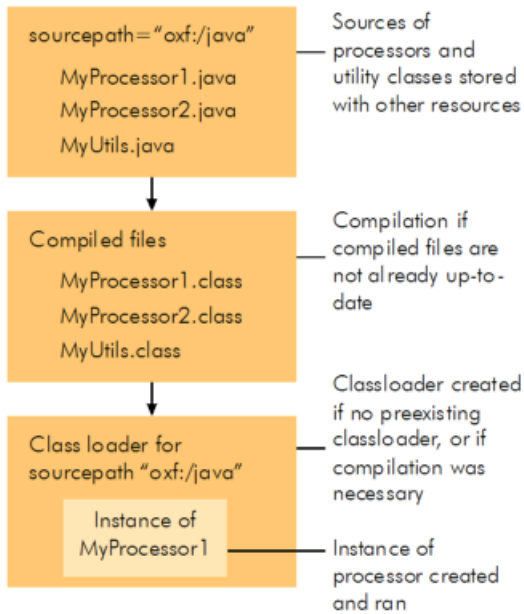
4.15.3. Usage



```
<p:processor name="oxf:java" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config sourcepath="oxf:/java" class="MyProcessor"/>
  </p:input>
  <p:input name="data" href="..." />
  <p:output name="data" id="..." />
</p:processor>
```

config input	<p>The <code>config</code> element has two attributes:</p> <ul style="list-style-type: none">■ The optional <code>sourcepath</code> attribute points to the directory containing the Java source in the resources. When omitted, the default <code>sourcepath</code> is the directory of the pipeline calling the Java processor (i.e. <code>sourcepath="."</code>). It is possible to use the <code>file:</code> protocol and the <code>oxf:</code> protocol. It is also possible to enter URLs relative to the location of the calling pipeline, such as <code>sourcepath="."</code> (the default), <code>sourcepath="../examples/java"</code>, etc.■ <code>class</code> is the name of the Java class. The class has to implement the <code>org.orbeon.oxf.processor.Processor</code> interface, as described in the Processors API. <p>Let's assume you place your Java source files in your resources directory under the <code>java</code> subdirectory, and that the class you want to use is <code>MyProcessor</code>, in the <code>com.example</code> package. Consequently, you will have a file <code>java/com/example/MyProcessor.java</code> in your resources. To use this class, the Java processor config is: <code><config sourcepath="oxf:/java" class="com.example.MyProcessor"/></code>.</p>
Other inputs and outputs	<p>The processor implemented in Java can take an arbitrary number of inputs and outputs. The only restriction on the inputs/outputs is that no input can be named <code>config</code> as this input is already used to configure the Java processor.</p>

4.15.4. Compilation



Before it can run a custom processor, the Java processor must compile the source code to generate the `class` files from the `java` files, and load those `class` files in the Java VM.

By default, the Java processor uses Sun's compiler (`com.sun.tools.javac.Main`) to produce `class` files. See the [compiler-class](#) and [compiler-jar](#) properties for more information about specifying the compiler to use.

The `class` files are stored in the temporary directory, as defined by the Java system property `java.io.tmpdir`. Since compilation is a time consuming process, it is performed only when necessary. The Java processor compiles a custom processor when one of these conditions is met:

- The source of the custom processor has never been compiled before.
- The last modified date of the source file for the custom processor is prior to the last modified date of the corresponding `class` file.

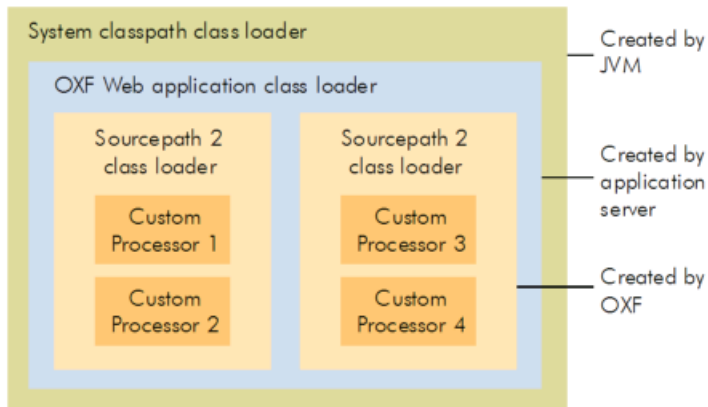
Note that Sun's `javac` automatically compiles all the files that the custom processor depends on, but that the Java processor only runs `javac` by comparing the dates of the `.java` and `class` of the custom processor itself. So if only one of the classes used by the custom processor has changed since the last compilation, the Java processor will not run the compiler. You should be aware of this limitation and "touch" the source of the custom processor when such a case occurs to force a compilation.

4.15.5. Compilation Class Path

Before invoking the Java compiler, OPS builds a classpath using two [properties](#). The following list summarizes the complete classpath order.

1. The class path defined by the [classpath](#) property.
2. The `WEB-INF/classes` directory, if found and Presentation Server runs in an application server.
3. The JAR and ZIP files used by the classloader hierarchy that loaded the Java processor. This automatically puts on the compilation class path the classes that the Java processor can use.
4. The JAR path defined by the [jarpath](#) property.
5. All the JAR and ZIP files under the `WEB-INF/lib` directory, if found and OPS runs in an application server.
6. If the `WEB-INF/lib` directory is not found, all the JAR and ZIP files in the same directory as the JAR file containing the Java processor. Typically, when running from a command line, this is the `ops.jar` JAR file. If `ops.jar` is stored in a directory with all the JAR files it depends on, those will automatically be added to the compilation class path.

4.15.6. Runtime Class Loading



The compiled files are loaded by a class loader created by the Java processor. A different class loader is created for each source path, and all the classes in the same source path are loaded in the same class loader. For a given source path, a new class loader is created if one of these conditions is met:

- No class loader has been previously created for this source path.
- One of the classes in this source path has been compiled since the class loader has been created.

When a new class loader is created, the previous one, if it exists, is discarded with all the loaded classes, and all the classes are re-loaded in the new class loader.

4.15.7. Limitations

- A custom processor used with the Java processor cannot use its `config` input, as this input is used to configure the Java processor.
- Java source files must be stored on the file system, i.e. the resources can only be loaded with the Filesystem or Web App [resource managers](#). This is due to a limitation of Sun's `javac` which can only compile source files stored on disk.
- The Java processor will recompile a custom processor only if the Java source of the processor itself has changed. If only one of the classes (that the custom processor depends on) has changed since the last compilation, the source of the customer processor must be "touched" to force a re-compilation.

4.15.8. Properties

Several global properties are relevant to the Java processor. Refer to the [Properties](#) section for more information.

4.15.9. Example

The processor below declares a single output `data` and no inputs. It will always send the same XML content to its `data` output, namely an `answer` element containing the text "42". For more details on how to implement processors in Java, please refer to the [Processors API](#).

```

import org.orbeon.oxf.pipeline.api.PipelineContext;
import org.orbeon.oxf.processor.ProcessorInputOutputInfo;
import org.orbeon.oxf.processor.SimpleProcessor;
import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.AttributesImpl;

public class DeepThoughtProcessor extends SimpleProcessor {

    public DeepThoughtProcessor() {
        addOutputInfo(new ProcessorInputOutputInfo(OUTPUT_DATA));
    }

    public void generateData(PipelineContext context,
                            ContentHandler contentHandler)
        throws SAXException {
        String answer = "42";
        contentHandler.startDocument();
        contentHandler.startElement("", "answer", "answer", new AttributesImpl());
        contentHandler.characters(answer.toCharArray(), 0, answer.length());
        contentHandler.endElement("", "answer", "answer");
        contentHandler.endDocument();
    }
}

```

4.16. Image Server

The Image Server processor serves images stored locally or remotely (for example through HTTP) to a Web browser. Only the JPEG format is supported at the moment. Before sending or transforming a resource, the Image Server checks that the resource is a JPEG image. The Image Server is able to perform simple transformations such as scaling and cropping, in which case it also handles a cache of transformed images.

4.16.1. Configuration

The `config` input must follow this Relax NG schema:

```

<element name="config" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <interleave>
    <element name="image-directory">
      <text/>
    </element>
    <element name="default-quality">
      <data type="float"/>
    </element>
    <optional>
      <element name="use-sandbox">
        <data type="boolean"/>
      </element>
    </optional>
    <optional>
      <element name="cache">
        <element name="directory">
          <text/>
        </element>
        <optional>
          <element name="path-encoding">
            <choice>
              <value>flat</value>
              <value>hierarchical</value>
            </choice>
          </element>
        </optional>
      </element>
    </optional>
  </interleave>
</element>

```

This is an example of configuration:

```

<config>
  <image-directory>file: C:/images</image-directory>
  <default-quality>0.8</default-quality>
  <cache>
    <directory>c:/oxf-image-cache</directory>
  </cache>
</config>

```

Element	Purpose	Format	Default
image-directory	Specifies the root of the directory containing all the images.	URL with a protocol specified. If the directory is local, use the file protocol. You can also use the http or oxf protocols.	None.
default-quality	Specifies the JPEG quality factor to use when encoding JPEG images..	Number between 0.0 and 1.0.	0.5
use-sandbox	<p>If set to false, it disables checking that the images served are strictly under the image-directory hierarchy.</p> <div> <p>Warning</p> <p>Disabling the sandbox can be a security hazard and should be used at your own risk. If the image paths come from untrustworthy sources, for example the URL entered by a user in a Web browser, you have to make sure that they do not access protected content. Ideally, only paths coming from trusted sources, such as your database or XML configuration files, should be used when the sandbox is disabled.</p> </div>	true or false.	true
cache	Optional element. If it is not specified, no caching of transformations takes place. If it is specified, at least the directory child element is required.	N/A	None.
cache/directory	Specifies the cache directory.	Path specifying the local filesystem directory that contains the cached transformed images.	None.
cache/path-encoding	Specifies how cache file names are computed. In this case, the cache builds a hierarchy of directories. A directory is created for each part of the image path separated by either a "/", a "\" or a ":". The benefit of this encoding is that in most cases, the cache directory hierarchy will mirror the hierarchy of the image directory. If different images can be accessed with paths differing only by the "/", a "\" or a ":", this scheme should not be used.		

If the `flat` cache path encoding scheme is selected, the cache will store all files directly under the cache directory. File names will be URL-encoded. This guaranties the uniqueness of the file names in the cache.

4.16.2. Image Input

Once the Image Server is configured, its `image` input can receive processing information. This input must follow this Relax NG schema:

```
<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <ref name="image"/>
  </start>
  <define name="image">
    <element name="image">
      <interleave>
        <element name="url">
          <text/>
        </element>
        <optional>
          <element name="quality">
            <data type="float"/>
          </element>
        </optional>
        <optional>
          <element name="use-cache">
            <data type="boolean"/>
          </element>
        </optional>
      </interleave>
      <zeroOrMore>
        <choice>
          <element name="transform">
            <attribute name="type">
              <value>scale</value>
            </attribute>
            <optional>
              <element name="quality">
                <choice>
                  <value>high</value>
                  <value>low</value>
                </choice>
              </element>
            </optional>
            <optional>
              <element name="scale-up">
                <data type="boolean"/>
              </element>
            </optional>
            <choice>
              <ref name="width-height"/>
              <ref name="max-size"/>
            </choice>
          </element>
          <element name="transform">
            <attribute name="type">
              <value>crop</value>
            </attribute>
            <interleave>
              <optional>
                <element name="x">
                  <data type="nonNegativeInteger"/>
                </element>
              </optional>
              <optional>
                <element name="y">
                  <data type="nonNegativeInteger"/>
                </element>
              </optional>
              <optional>
                <element name="width">
                  <data type="positiveInteger"/>
                </element>
              </optional>
              <optional>
                <element name="height">
                  <data type="positiveInteger"/>
                </element>
              </optional>
            </interleave>
          </element>
          <element name="transform">
            <attribute name="type">
              <value>draw</value>
            </attribute>
          </element>
        </choice>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

```

</attribute>
<oneOrMore>
  <choice>
    <element name="rect">
      <attribute name="x">
        <data type="nonNegativeInteger"/>
      </attribute>
      <attribute name="y">
        <data type="nonNegativeInteger"/>
      </attribute>
      <attribute name="width">
        <data type="positiveInteger"/>
      </attribute>
      <attribute name="height">
        <data type="positiveInteger"/>
      </attribute>
      <optional>
        <ref name="color"/>
      </optional>
    </element>
    <element name="fill">
      <attribute name="x">
        <data type="nonNegativeInteger"/>
      </attribute>
      <attribute name="y">
        <data type="nonNegativeInteger"/>
      </attribute>
      <attribute name="width">
        <data type="positiveInteger"/>
      </attribute>
      <attribute name="height">
        <data type="positiveInteger"/>
      </attribute>
      <optional>
        <ref name="color"/>
      </optional>
    </element>
    <element name="line">
      <attribute name="x1">
        <data type="int"/>
      </attribute>
      <attribute name="y1">
        <data type="int"/>
      </attribute>
      <attribute name="x2">
        <data type="int"/>
      </attribute>
      <attribute name="y2">
        <data type="int"/>
      </attribute>
      <optional>
        <ref name="color"/>
      </optional>
    </element>
  </choice>
</oneOrMore>
</element>
</choice>
</zeroOrMore>
</element>
</define>
<define name="width-height">
  <interleave>
    <element name="width">
      <data type="positiveInteger"/>
    </element>
    <element name="height">
      <data type="positiveInteger"/>
    </element>
  </interleave>
</define>
<define name="max-size">
  <choice>
    <element name="max-size">
      <data type="positiveInteger"/>
    </element>
    <element name="max-width">
      <data type="positiveInteger"/>
    </element>
    <element name="max-height">
      <data type="positiveInteger"/>
    </element>
  </choice>
</define>
<define name="color">
  <element name="color">
    <choice>

```

```

    <attribute name="name">
      <choice>
        <value>white</value>
        <value>lightGray</value>
        <value>gray</value>
        <value>darkGray</value>
        <value>black</value>
        <value>red</value>
        <value>pink</value>
        <value>orange</value>
        <value>yellow</value>
        <value>green</value>
        <value>magenta</value>
        <value>cyan</value>
        <value>blue</value>
      </choice>
    </attribute>
    <attribute name="rgb">
      <data type="string">
        <param name="pattern">#[0-9A-Fa-f]{6}</param>
      </data>
    </attribute>
  </choice>
</optional>
  <attribute name="alpha"/>
</optional>
</element>
</define>
</grammar>

```

The only required element is the `url` element. This is interpreted as a URL relative to the image directory.

If `use-sandbox` is not set to `false` and the resulting path is not in the sandbox, the processor returns a 404 error code to the Web browser. If the resource does not exist, the processor also returns a 404 error code to the Web browser.

The cache can be disabled by setting the `use-cache` element to `false`. It defaults to `true`.

If only the `url` element is set, no transformation takes place.

Zero or more transformations can be specified with the `transform` element. Each transformation is identified by a `type` attribute that identifies the type of transformation. Each transformation is performed sequentially. If at least one transformation is specified, the `quality` element can be used to override the configuration's default JPEG quality setting.

If the `type` attribute is set to `scale`, a scaling operation is performed. It is possible to either specify a width and height to scale the image to, or one of `max-size`, `max-width` or `max-height`. If `scale-up` is set to `false`, no scaling takes place if the specified parameters produce an image larger than the original image.

The `quality` element can be set to `low` to use a faster but lower-quality algorithm. The default is `high`.

If the `type` attribute is set to `crop`, a cropping operation is performed. All parameters are optional: `x`, `y`, `width` and `height`. `x` and `y` specify the top left corner of the cropping rectangle. They default to 0. `width` and `height` specify the size of the cropping rectangle. They default to covering the rest of the image to the right and bottom sides.

Example of image input that will make sure that the maximum size of the image is 100 pixels, without scaling up:

```

<image>
  <url>relative-path/to/my/image.jpg</url>
  <transform type="scale">
    <scale-up>false</scale-up>
    <max-size>100</max-size>
  </transform>
</image>

```

Example of use of the Image Server processor:

```

<p:processor name="oxf:image-server" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="image">
    <image>
      <url>relative-path/to/my/image.jpg</url>
      <transform type="scale">
        <scale-up>false</scale-up>
        <max-size>100</max-size>
      </transform>
    </image>
  </p:input>
  <p:input name="config">
    <config>
      <image-directory>file:C:/images</image-directory>
      <default-quality>0.8</default-quality>
      <cache>
        <directory>c:/oxf-image-cache</directory>
      </cache>
    </config>
  </p:input>
</p:processor>

```

Warning

Image transformations can take a lot of memory depending on the size of the source and transformed images. Be sure to set your memory settings accordingly. Concurrent transformations can also yield to higher memory consumption.

4.16.3. Drawing

The Image Server also supports drawing basic shapes on an image. Empty and filled rectangles, and lines are supported. Each shape may have a color element.

```
<i image>
  <url>ca-coast.jpg</url>
  <quality>0.7</quality>
  <use-cache>false</use-cache>
  <rect x="10" y="10" height="100" width="100">
    <color rgb="#ff0000" alpha="ff"/>
  </rect>
  <fill x="100" y="100" height="200" width="200">
    <color rgb="#00ff00" alpha="55"/>
  </fill>
  <line x1="200" y1="200" x2="300" y2="300">
    <color rgb="#0000ff" alpha="ff"/>
  </line>
</i image>
```

4.17. Charts and Spreadsheets

4.17.1. Chart Processor

OPS uses the [JFreeChart](#) to draw and display charts. The following features are supported:

- Bar Chart
- Stacked Bar Chart
- Line Chart
- Pie Chart
- Area Chart
- Customizable size, colors, title and legend
- Output HTML image map and tool tips

A chart consists of two axes. The horizontal (X) axis is called the *category* axis. The vertical (Y) axis is called the *value* axis. The chart contains one or more *values*. Each value is a list of number to be charted. A value entry contains two lists:

- the categories list, represented in the X axis
- the series list, represented on the Y axis

The lists of value are created with XPath expressions evaluated against the XML document in the *data* input. These expressions must return a node set, and every expression must return a node set of the same length.

The *chart* input contains the configuration information for the processor. The following table shows elements that are available inside the *chart* root element. The [Chart Example](#) shows most of these options.

Note

The colors are always specified in RGB format, prefixed by a #. For instance, #FF0000 is a pure red.

type	The chart type. The following value are supported vertical-bar; horizontal-bar; vertical-bar-3d; horizontal-bar-3d; stacked-vertical-bar; stacked-horizontal-bar; stacked-vertical-bar-3d; stacked-horizontal-bar-3d; line; area; pie; pie-3d
title	The title, printed on top of the chart.
title-color	The font title color
background-color	Color of the background of the chart
x-size	The horizontal size of the chart image, in pixels
y-size	The vertical size of the chart image, in pixels
category-title	The label printed on the horizontal (X) axis
category-margin	Margin between two categories, in percentage of width. Applies only to the vertical-bar chart type.
serie-title	The label printed on the vertical (Y) axis
map	Name of the HTML image map
bar-margin	Margin between two vertical bars, in percentage of width. Applies only to the vertical-bar chart type.

tick-unit	The vertical (Y) axis unit
category-label-angle	Controls the angle of the category axis labels. This value is a positive angle in degree.
legend	Controls the legend box.
legend/@visible	Control if the legend is displayed. Can be true or false
legend/@position	The position of the legend relative to the chart. Possible values are east, north, west or south
legend/item	Forces manual legend. You can specify any number of items, each with a label and color attribute.
value	Defines the values of the chart. At least one element must be present
value/@title	The name of the value item
value/@categories	XPath expression returning the category list
value/@series	XPath expression returning the serie list
value/@colors	XPath expression returning a color list
exploded-percent	XPath expression returning a percentage list. Valid only for Pie charts and allows certain slices to be exploded from the pie.
value/color	Manual override of the color for this value

The full RelaxNG schema is shown below:

```
<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <ref name="chart"/>
  </start>
  <define name="chart">
    <element name="chart">
      <interleave>
        <element name="type">
          <choice>
            <value>vertical-bar</value>
            <value>horizontal-bar</value>
            <value>vertical-bar-3d</value>
            <value>horizontal-bar-3d</value>
            <value>stacked-vertical-bar</value>
            <value>stacked-horizontal-bar</value>
            <value>stacked-vertical-bar-3d</value>
            <value>stacked-horizontal-bar-3d</value>
            <value>line</value>
            <value>area</value>
            <value>pie</value>
            <value>pie-3d</value>
          </choice>
        </element>
        <element name="title">
          <data type="string"/>
        </element>
        <optional>
          <element name="map">
            <data type="string"/>
          </element>
        </optional>
        <optional>
          <element name="category-title">
            <data type="string"/>
          </element>
        </optional>
        <optional>
          <element name="category-margin">
            <data type="double"/>
          </element>
        </optional>
        <optional>
          <element name="serie-title">
            <data type="string"/>
          </element>
        </optional>
        <oneOrMore>
          <element name="value">
            <attribute name="title"/>
            <attribute name="categories"/>
            <attribute name="series"/>
            <optional>
              <attribute name="colors"/>
            </optional>
          </element>
        </oneOrMore>
      </interleave>
    </element>
  </define>
</grammar>
```

```

        <optional>
          <attribute name="exploded-percents"/>
        </optional>
        <optional>
          <attribute name="color">
            <ref name="color"/>
          </attribute>
        </optional>
        <empty/>
      </element>
    </oneOrMore>
    <element name="x-size">
      <data type="integer"/>
    </element>
    <element name="y-size">
      <data type="integer"/>
    </element>
    <optional>
      <element name="background-color">
        <ref name="color"/>
      </element>
    </optional>
    <optional>
      <element name="title-color">
        <ref name="color"/>
      </element>
    </optional>
    <optional>
      <element name="bar-margin">
        <data type="double"/>
      </element>
    </optional>
    <optional>
      <element name="tick-unit">
        <data type="double"/>
      </element>
    </optional>
    <optional>
      <element name="category-label-angle">
        <data type="positiveinteger"/>
      </element>
    </optional>
    <optional>
      <element name="legend">
        <optional>
          <attribute name="visible">
            <data type="boolean"/>
          </attribute>
        </optional>
        <optional>
          <attribute name="position">
            <choice>
              <value>north</value>
              <value>east</value>
              <value>south</value>
              <value>west</value>
            </choice>
          </attribute>
        </optional>
        <zeroOrMore>
          <element name="item">
            <attribute name="color">
              <ref name="color"/>
            </attribute>
            <attribute name="label">
              <data type="string"/>
            </attribute>
            <empty/>
          </element>
        </zeroOrMore>
      </element>
    </optional>
  </interleave>
</element>
</define>
<define name="color">
  <data type="string">
    <param name="pattern">#[0-9A-Fa-f]{6}</param>
  </data>
</define>
</grammar>

```

The Chart Serializer outputs an XML document on its `data` output, which describes the rendered chart information. It contains the chart image name and an optional HTML image map. A typical document looks like below. The `file` element contains the unique name of the generated chart. The `map` element contains the image map.

```

<chart-info>
  <file>jfreechart-20234.png</file>
  <map name="#fruits">
    <area shape="RECT" coords="66,54,86,235" title="May, Apples = 10"/>
    <area shape="RECT" coords="93,35,113,234" title="June, Apples = 11"/>
    <area shape="RECT" coords="131,72,151,235" title="May, Oranges = 9"/>
    <area shape="RECT" coords="158,90,178,235" title="June, Oranges = 8"/>
    <area shape="RECT" coords="196,90,216,235" title="May, Bananas = 8"/>
    <area shape="RECT" coords="223,126,243,235" title="June, Bananas = 6"/>
    <area shape="RECT" coords="261,108,281,235" title="May, Berries = 7"/>
    <area shape="RECT" coords="289,54,309,235" title="June, Berries = 10"/>
    <area shape="RECT" coords="326,126,346,235" title="May, Pears = 6"/>
    <area shape="RECT" coords="354,163,374,235" title="June, Pears = 4"/>
  </map>
</chart-info>

```

You need to setup a special servlet in your Web application descriptor (web.xml) to serve the chart image file. The following line declares the servlet and maps it to /chartDisplay

```

<servlet>
  <servlet-name>DisplayChart</servlet-name>
  <servlet-class>com.jrefinery.chart.servlet.DisplayChart</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>DisplayChart</servlet-name>
  <url-pattern>/chartDisplay</url-pattern>
</servlet-mapping>

```

The following XSLT template can be used to extract the chart information and generate HTML:

```

<xsl:template match="chart-info" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  
  <xsl:copy-of select="map"/>
</xsl:template>

```

4.17.2. Excel Processors

Note

These processors are deprecated. Please refer to the [Converters](#) section for up-to-date information.

OPS ships with the [POI](#) library which allows import and export of Microsoft Excel documents. OPS uses an Excel file template to define the layout of the spreadsheet. You define cells that will contain the values with a special markup.

First, create an Excel spreadsheet with the formatting of your choosing. Values are taken from the data input document with an XPath expression. Apply a special markup to the cell you need to export values to:

1. Select the cell
2. Go to the menu **Format->Cell**
3. In the **Number** tab, choose the **Custom** format and enter a format that looks like: `#,##0;" /a/b" /c/d`. In this example we have 2 XPath expressions separated by a pipe character (`|`): `/a/b` and `/c/d`. The first XPath expression is used when creating the Excel file (exporting) and is run against the data input document of the XLS Serializer processor (see below). The second expression is optional and is used when recreating an XML document from the Excel file (importing).

The XLS Serializer processor takes a `config` input describing the XLS template file, and a `data` input containing the values to be inserted in the template. The processor scans the template, and applies XPath expressions to fill in the template. The `config` input takes a single element with two attributes:

template	A URL pointing to an XLS template file
filename	The name under which the file is saved.

```

<p:processor name="oxf:xls-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config template="oxf:/excel/template.xls" filename="currency.xls"/>
  </p:input>
  <p:input name="data">
    <currency>
      <val ue1>10</val ue1>
      <val ue2>20</val ue2>
      <val ue3>30</val ue3>
    </currency>
  </p:input>
</p:processor>

```

```
<form action="excel/import" method="post" enctype="multipart/form-data">
  <input type="file" name="xls"/>
  <input type="submit" value="Import"/>
</form>
```

The XPL pipeline can then instantiate the XLS Generator with the following:

```
<p:processor name="oxf:xls-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="request" href="#request"/>
  <p:output name="data" id="xls"/>
</p:processor>
```

4.18. PDF Extraction Processor

4.18.1. Rationale

The PDF extraction processor will take an PDF file as input and extract meta information and or text from the PDF. The extracted structure can be used by fulltext search or a content management system to identify the exact location of content items. The processor makes use of the PDFBox Library [PDFBox on Sourceforge](#).

The processor allows 5 modes of operation. In mode one only the meta data and the bookmarks with title and page number but no text is extracted. In mode two meta data, bookmarks and text is extracted. In mode three meta data and text broken down into pages is extracted. In mode 4 only the meta data gets extracted (if any). In mode five first an extraction of bookmarks with text is attempted. If there are no bookmarks a fallback to pages is performed.

If the file contains errors, then the operation might not complete. Most errors are captured and will lead to the insertion of an <error> tag. If the input file is fundamentally broken no output will be displayed (however the processor returns an empty <PDFDocument /> entry.

Note

This processor is a contribution from TAO Consulting Pte Ltd.

4.18.2. Usage

Processor Name	tao:from-pdf-converter
config input	Definition of the scope of extraction.
data input	The PDF document in Base64 encoding
data output	The XML Structure extracted from the PDF

The configuration input selects the mode of the extraction. Possible keywords are: *bookmarks*, *bookmarksonly*, *bookmarkspages*, *meta* or *pages*. Depending on that the extraction takes place. "bookmarkspages" attempts to extract bookmarks with text enclosed. If the PDF doesn't contain bookmarks the processor falls back to extract text by page.

```
<config>
  <action>bookmarks</action>
</config>
```

The data input must contain the PDF converted to Base64. This can happen from URL Generator or the x-forms upload control. The Base64 encoding must comply to the [binary document format](#).

```
<p:input name="data" href="#file" xmlns:p="http://www.orbeon.com/oxf/pipeline"/>
```

The data output is a XML structure with all PDF specific information. It starts with a PDFDocument element followed by the PDFMetadata element that contains PDF meta data according to the [Adobe PDF](#) specification. The document is then followed by either Bookmark or Page Elements

For example, the following element could be generated:

```
<PDFDocument pages="32" author="Stephan H. Wissel" title="R6 Migration Report" subject="Recommendation for Migration">
  <PDFMetadata>... a lot of stuff here ...</PDFMetadata>
  <Bookmark level="1" page="2">
    <Title>Management Summary</Title>
  </Bookmark>
  <Bookmark level="1" page="3">
    <Title>Scope of work / Findings</Title>
    <Bookmark level="2" page="3">
      <Title>Scope of work</Title>
    </Bookmark>
  </Bookmark>
</PDFDocument>
```

4.19. Email Processor

4.19.1. Scope

The Email processor can send emails through an SMTP server (the usual way of sending emails). Its input contains the basic configuration (SMTP host, subject, etc.) as well as inline message body content. Alternatively, the message content can refer to external resources, such as resources on disk or dynamically generated content. It features the following high-level functionality:

- **Multiple recipients:** send the same email to multiple recipients.
- **Multipart hierarchy:** it is possible to have multiple levels of multipart messages.
- **Binary attachments:** binary files such as images and PDF files can be attached to an email.
- **Dynamic attachments:** attachments can be generated dynamically. It is for example possible to attach a dynamically generated chart or PDF file.

4.19.2. Data Input

The data input contains the configuration of the processor as well as the message header and body. The following table describes the configuration elements:

Name	Cardinality	Description
message	1	Root element
message/smtp-host	0..1	The SMTP host used to send the message
message/from	1	Sender of the message. Contains an <code>email</code> element and an optional <code>name</code> element.
message/to	1..n	Recipient(s) of the message. Contains an <code>email</code> element and an optional <code>name</code> element.
message/cc	0..n	Carbon copy recipient(s) of the message. Contains an <code>email</code> element and an optional <code>name</code> element.
message/bcc	0..n	Blind carbon copy recipient(s) of the message. Contains an <code>email</code> element and an optional <code>name</code> element.
message/subject	1	Subject of the message
message/header	0..n	Optional extra email header to add. Contains a <code>name</code> element and a <code>value</code> element.
message/body	1	Indicates a message body optionally containing multiple parts
message/body/@content-type	1	The content-type of this body part. This attribute can also include a <code>charset</code> attribute to specify a character encoding for text types. For example: <code>text/plain; charset=utf-8</code> . This attribute may also specify a multipart data type: <code>multipart/mixed</code> , <code>multipart/alternative</code> or <code>multipart/related</code> .
message/body/part	0..n	A message body part, if the body element specifies a multipart <code>content-type</code> attribute.
message/body/part/@name	1	The name of this body part
message/body/part@content-type	1	The content-type of this body part. This can also include a <code>charset</code> attribute to specify a character encoding for text types. For example: <code>text/plain; charset=utf-8</code> . This attribute may also specify a multipart data type: <code>multipart/mixed</code> , <code>multipart/alternative</code> or <code>multipart/related</code> . In this case, the part contains an embedded multipart message. This replaces the deprecated <code>mime-multipart</code> attribute.
message/body/part@content-disposition	0..1	The optional Content-Disposition header of this body part. Not allowed if the part contains embedded parts.
message/body/part@content-id	0..1	The optional Content-ID header of this body part.
message/body/part/*	1	The content of the body part. This can contain embedded <code>part</code> elements if the content is multipart. It can be XHTML if the content-type is <code>text/html</code> . Finally, it can be any text content, including just plain HTML (which can be embedded in a CDATA section for convenience).

4.19.3. Simple Messages

A simple message requires a `body` element with:

- A `text content-type` attribute, for example `text/plain`
- Text content

For example:

```
<p:processor name="oxf:email" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data">
    <message>
      <smtp-host>mail.example.org</smtp-host>
      <from>
        <email>trove@smith.com</email>
        <name>Trove Smith</name>
      </from>
```

```

    <to>
      <email>jani@smith.com</email>
      <name>Jani Smith</name>
    </to>
    <subject>Reminder</subject>
    <body content-type="text/plain">
      Hello, Jani!
    </body>
  </message>
</p:input>
</p:processor>

```

4.19.4. Character Encoding

In the example above, no character encoding is specified for the `body` element. This determines what character encoding is used in the body of the email message constructed by the Email processor. If no encoding is specified, the default `iso-8859-1` is used. In some cases, it is useful to specify a character encoding. For example, if it is known that the message only contains ASCII characters, the `us-ascii` encoding can be specified. If, on the other hand, the message contains characters from multiple languages, the `utf-8` encoding can be specified.

Use the `content-type` attribute to specify an encoding, for example: `content-type="text/plain; charset=utf-8"`.

Note

XML itself support Unicode, in other words it is designed to allow representing all the characters specified by the Unicode specification. Those characters can all be represented with the UTF-8 encoding. Because of this, the Email processor could always encode text using the UTF-8 encoding. However, some mail clients may not all support that encoding. It is therefore left to the user of the Email processor to specify the appropriate encoding.

4.19.5. Message Parts

An email message can be composed of several parts. Parts can be used for:

- **Attachments:** for example, a simple text message may have one or more image attachments. Usually, the `multipart/mixed` content type is used for this purpose.
- **Alternative Formats:** for example, both a plain text and an HTML version of a same message may be sent. The recipient, or her mail software, can then choose the most appropriate format to display. The `multipart/alternative` content type is used for this purpose.
- **Dependencies:** for example, an HTML message may refer to images or other resources embedded in the same email. The `multipart/related` content type is used for this purpose.

To create a multipart email, specify one of the multipart content types on the `body` element. The `body` element must contain one or more `part` elements.

In turn, `part` elements may contain other parts. In that case, a `part` element must declare a multipart content type attribute, and contain at least one `part` element.

The main part of the body is encapsulated by the `body` element of the message.

4.19.6. Inline and Out of Line Parts

The content of a part can be specified in two ways:

- **Inline:** the content is directly embedded in the `body` or `part` element.
- **Out of line:** the content is available from a resource or dynamically generated.

The content of the `body` or `part` element can be of the following types:

- **HTML:** the content type is `text/html`. In this case, the inline content is considered as HTML and converted to HTML. A root `html` element must be present.
- **Text type:** this is the case when the content type starts with `text/`, for example `text/plain`. In this case, a character encoding can be specified as well.
- **Binary Type:** for all other content types, the body of the part must contain Base64-encoded binary data.

This mode is enabled when the `part` element contains an `src` attribute.

You can refer to a part content using a regular URI, for example:

```
<part src="oxf:/image.jpg" content-type="image/jpeg"/>
```

You can also refer to dynamically generated content by referring to optional processor inputs. For example:

```
<part src="input:image-content" content-type="image/jpeg"/>
```

In this case, the content of the image is obtained by reading the `image-content` input of the Email processor. You can choose an arbitrary name for the input, as long as it is not `data`. Then, connect a processor to the input, for example:

```

<p:processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <url>oxf:/image.jpg</url>
      <content-type>image/jpeg</content-type>
    </config>
  </p:input>
  <p:output name="data" id="file"/>
</p:processor>
<p:processor name="oxf:email" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data">
    <message>
      ...
    </message>
  </p:input>
  <p:input name="image-content" href="#file"/>
</p:processor>

```

When the content type of the input read is text (starts with `text/`) or XML (`text/xml`, `application/xml`, or ends with `+xml`), the input document is expected to be in the text format specified by the [text document format](#). When the content type of the input read is binary (all the other cases), the input document is expected to be in the binary format specified by the [binary document format](#).

4.19.7. Properties

Several global properties are relevant to the Email processor. Refer to the [Properties](#) section for more information.

4.19.8. Examples

This example shows how to send both a text and HTML version of a message to two recipients.

```

<p:processor name="oxf:email" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data">
    <message>
      <smtp-host>mail.company.com</smtp-host>
      <from>
        <email>trove@smith.com</email>
        <name>Trove Smith</name>
      </from>
      <to>
        <email>jani@smith.com</email>
        <name>Jani Smith</name>
      </to>
      <to>
        <email>tori@smith.com</email>
        <name>Tori Smith</name>
      </to>
      <subject>Reminder</subject>
      <body mime-multi-part="alternative">
        <part name="part1" content-type="text/plain">
          This is part 1
        </part>
        <part name="part2" content-type="text/html">
          <html>
            <body>
              <p>
                This is part 2
              </p>
            </body>
          </html>
        </part>
      </body>
    </message>
  </p:input>
</p:processor>

```

This example shows how to send related parts with HTML, as well as dynamically generated attachments.

```

<p:processor name="oxf:email" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="data">
    <message>
      <smtp-host>mail.company.com</smtp-host>
      <from>
        <email>trove@smith.com</email>
        <name>Trove Smith</name>
      </from>
      <to>
        <email>jani@smith.com</email>
        <name>Jani Smith</name>
      </to>
      <subject>Email Example</subject>
      <body mime-multi-part="alternative">

```

```

<!-- Provide simple text alternative -->
<part name="text" content-type="text/plain">This is some important message body.</part>
<!-- HTML alternative -->
<part name="html" content-type="multipart/related">
  <part name="main" content-type="text/html">
    <html>
      <head>
        <title>Email Example</title>
      </head>
      <body>
        <p style="border: dotted 1px gray; padding: 5px">
          This is some<em>important</em>message body.
        </p>
        <p>
          This is a static image attached to the email and referred to by the HTML version:
        </p>
        <div style="border: dotted 1px gray; padding: 5px">
          
        </div>
        <p>
          This is an dynamic chart image attached to the email and referred to by the HTML
          version:
        </p>
        <div style="border: dotted 1px gray; padding: 5px">
          
        </div>
      </body>
    </html>
  </part>
  <!-- Attachments -->
  <part name="image" content-type="image/gif" content-disposition="inline; filename="logo.gif"
content-id="id1" src="oxf:/logo.gif"/>
  <part name="chart" content-type="image/png" content-disposition="inline; filename="chart.png"
content-id="id2" src="input:png-document"/>
  <part name="pdf" content-type="application/pdf" content-disposition="inline;
filename="report.pdf" src="input:pdf-document"/>
</part>
</body>
</message>
</p:input>
<p:input name="png-document"/>
<p:input name="pdf-document"/>
</p:processor>

```

4.20. Yahoo Instant Messaging Processor

4.20.1. Scope

This processor can send and receive Yahoo Instant Messages (IM) from within a Web application. To use this processor, you need to create a Yahoo account that will act as the sender. You can send an IM to any Yahoo subscriber. The processor launches a customizable pipeline when an IM is received. The IM processor has two mandatory inputs: `session` contains login information, and `message` contains the message to send. A third input, declared in the `on-message-received` element contains the pipeline to be executed when a message is received. This pipeline must have a `data` input parameter containing a document describing the incoming message:

```

<message>
  <from>john</from>
  <body>Hello!</body>
</message>

```

4.20.2. Config Input

The `config` input contains basic information such as login, password and definition of the pipeline to execute when an IM is received.

login	A Yahoo account, used as the sender of the IM
password	The matching password
on-message-received	<ul style="list-style-type: none"> A URL pointing to an XPL pipeline An input name, starting with a #. The <code>IMSerializer</code> must have the corresponding input declared.

4.20.3. Data Input

The `data` input contains the message to send under the `message` element.

to	A Yahoo account to send to message to
body	The body of the message to send

4.20.4. Example

The following code illustrates the instantiation of the IMSerializer to send a message to the Yahoo account tove from jani. Additionally, a statement is written in the log file when a message is received.

```
<p: processor name="oxf:im" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <session>
      <login>jani</login>
      <password>secret</password>
      <on-message-received>#response-pipeline</on-message-received>
    </session>
  </p:input>
  <p:input name="data">
    <message>
      <to>trove</to>
      <body>Don't forget me!</body>
    </message>
  </p:input>
  <p:input name="response-pipeline">
    <p:config>
      <p:param name="data" type="input"/>
      <p:processor name="oxf:null-serializer">
        <p:input name="data" href="#data"/>
      </p:processor>
    </p:config>
  </p:input>
</p:processor>
```

4.21. Pipeline Processor

4.21.1. Introduction

The Pipeline processor provides support for sub-pipelines in XPL. It allows XPL programs to be used and manipulated like XML processors. It allows building the equivalent of functions and procedures in other programming languages.

4.21.2. Inputs and Outputs

Type	Name	Purpose	Mandatory
Input	config	Configuration following the XPL syntax	Yes
Input	User-defined inputs	User-defined inputs accessible from the called XPL pipeline with <code>p:param</code> .	No
Output	User-defined outputs	User-defined outputs accessible from the called XPL pipeline with <code>p:param</code> .	No

4.21.3. Example of Use

Like any XML processor, the Pipeline processor is used from within an XPL pipeline. It is important to make a distinction between the *calling XPL pipeline*, which is the pipeline making use of the Pipeline processor, and the *called XPL pipeline*, which is the sub-pipeline called by the calling pipeline. This is similar to other programming language where, for example, a function calls another function.

This is an example of using the Pipeline processor in a calling XPL pipeline to call a sub-pipeline:

```
<p: processor name="oxf:pipeline" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <!-- The config input provides the declaration of the sub-pipeline -->
  <p:input name="config" href="get-categories.xpl"/>
  <!-- The query input is an optional input that allows passing an XML document to the query input of the sub-pipeline -->
  <p:input name="query">
    <query>
      <username>j doe</username>
      <blog-id>123456</blog-id>
    </query>
  </p:input>
  <!-- The categories output is an optional output that allows reading the categories output of the sub-pipeline -->
  <p:output name="categories" id="resulting-categories"/>
</p:processor>
```

Note

In this example the `config` input is contained in a separate resource file called `get-categories.xpl`. Because XPL configurations are usually fairly long, this is often the preferred way of referring to a pipeline. But as usual with XPL, it is also possible to inline the content of inputs (as the `query` input above shows), or to refer to a dynamically-generated XML document.

4.21.4. Configuration

The config input must follow the XPL syntax. For example the `get-categories.xpl` sub-pipeline in the example above can contain:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:param type="input" name="query" schema-href="get-categories-query.rng"/>
  <p:param type="output" name="categories" schema-href="get-categories-result.rng"/>
  <p:processor name="oxf:xslt">
    <p:input name="data" href="#query"/>
    <p:input name="config">
      <xdb:query collection="/db/orbeon/blog-example/blogs" create-collection="true" xsl:version="2.0"
        xmlns:xdb="http://orbeon.org/oxf/xml/xml-db">
        xquery version "1.0";
        <categories>
          { for $i in (/blog[username = '
            <xsl:value-of select="/query/username" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
            ' and blog-id = '
            <xsl:value-of select="/query/blog-id" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
            ' ])[1]/categories/category return
            <category>
              <name>{xs:string($i/name)}</name>
              <id>{count($i/preceding-sibling::category) + 1}</id>
            </category>
          }
        </categories>
      </xdb:query>
    </p:input>
    <p:output name="data" id="xml-db-query"/>
  </p:processor>
  <p:processor name="oxf:xml-db-query">
    <p:input name="datasource" href="../datasource.xml"/>
    <p:input name="query" href="#xml-db-query"/>
    <p:output name="data" ref="categories"/>
  </p:processor>
</p:config>
```

4.21.5. Optional Inputs

Optional inputs allow passing XML documents to sub-pipeline. The example sub-pipeline above requires a `query` input:

```
<p:param type="input" name="query" schema-href="get-categories-query.rng" xmlns:p="http://www.orbeon.com/oxf/pipeline"/>
```

The caller must therefore pass an XML document that will be received by the sub-pipeline. This is done by setting, on the Pipeline processor, an input with name `query`:

```
<p:processor name="oxf:pipeline" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="query">...</p:input>
</p:processor>
```

Note

Because the `config` input of the Pipeline processor is used to provide the XPL pipeline, a sub-pipeline should not expose a `config` input, as it won't be possible to connect that input.

Other inputs can similarly be passed to the sub-pipeline, simply by connecting inputs on the Pipeline processor that match the names of the inputs declared by the sub-processor.

4.21.6. Optional Outputs

Optional outputs allow retrieving XML documents produced by a sub-pipeline. The example sub-pipeline above requires a `categories` output:

```
<p:param type="output" name="categories" schema-href="get-categories-result.rng" xmlns:p="http://www.orbeon.com/oxf/pipeline"/>
```

The caller, in order to read the `categories` output of the sub-pipeline, must set, on the Pipeline processor, an output with name `categories`:

```
<p:processor name="oxf:pipeline" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:output name="categories" id="resulting-categories"/>
</p:processor>
```

As usual with XPL, the `id` attribute provided on the output is chosen by the user. Here, we use a `resulting-categories` identifier.

Other outputs of the sub-pipeline can similarly be read, simply by connecting outputs on the Pipeline processor that match the names of the outputs declared by the sub-processor.

4.22. Validation Processor

4.22.1. Rationale

The validation processor can be inserted in a pipeline to validate data against a specified schema. The current implementation supports [W3C Schema](#) and [Relax NG](#).

The validator functions in two distinct mode. The *decorating* mode adds attribute to the document where errors occur. The *non-decorating* mode throws a `org.orbeon.oxf.common.ValidationException` if the data doesn't conform to the schema.

4.22.2. Usage

Processor Name	oxf.validation
config input	The configuration of this validator.
schema input	The schema (W3C Schema or Relax NG).
data input	The document to validate.
data output	This output mirrors the data input.

The configuration input selects the mode of the validator. The validator can either be in the *decorating* mode or the *non-decorating* mode. The *decorating* element contains a boolean (`true` or `false`) indicating if the validator is decorating or not. The following example shows a configuration for a decorating validator.

```
<config>
  <decorate>true</decorate>
</config>
```

The validation processor's schema input contains a supported XML schema (W3C or Relax NG). The schema type is automatically recognized. The following example shows a simple Relax NG schema for an arbitrary address book.

```
<element name="addressBook">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <attribute name="age">
          <text/>
        </attribute>
      <text/>
    </element>
    <element name="email">
      <text/>
    </element>
  </zeroOrMore>
</element>
```

For more information about Relax NG syntax, read the [Relax NG specification](#) and [Relax NG tutorial](#). The [W3C Schema Primer](#) provides a good introduction to the language.

The data input contains the xml data to be validated. The following document is valid against the address book schema defined above.

```
<addressBook>
  <card>
    <name age="24">John Smith</name>
    <email>js@example.com</email>
  </card>
  <card>
    <name age="42">Fred Bloggs</name>
    <email>fb@example.net</email>
  </card>
</addressBook>
```

If the input data is valid against the specified schema, this output mirrors the input data, i.e. the validation processor is invisible in the pipeline. However, when validation errors occur, a `ValidationException` is thrown and the pipeline processing is interrupted if the validator is in the *non-decorating* mode. When in *decorating* mode, the validator annotates the output document in the following way:

For each validation error, the validator inserts an additional element after the error-causing element. This element is in the `http://orbeon.org/oxf/xml/validation` namespace URI and contains the following information:

- The message of the validator
- The system ID of the document, if available
- The location (line and column) within the document, if available.

For example, the following element could be generated:

```
<v:error message="Error bad character content for element near month (schema: oxf:/date.rng)" system-id="oxf:/date.xml" line="5" column="10" xmlns:v="http://orbeon.org/oxf/xml/validation"/>
```

4.23. Scheduler Processor

4.23.1. Scheduler

The OPS Scheduler allows applications to start and stop periodic tasks. A OPS task is defined by a OPS processor and its inputs. You can start or stop a task at any time. The `config` input must conform to the following schema:

```
<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <ref name="config"/>
  </start>
  <define name="config">
    <element name="config">
      <interleave>
        <zeroOrMore>
          <element name="start-task">
            <interleave>
              <element name="name">
                <data type="string"/>
              </element>
              <choice>
                <element name="processor-name">
                  <data type="QName"/>
                </element>
              </choice>
              <element name="start-time">
                <data type="string"/>
              </element>
              <element name="interval">
                <data type="long"/>
              </element>
              <optional>
                <element name="synchronised">
                  <data type="boolean"/>
                </element>
              </optional>
              <zeroOrMore>
                <element name="input">
                  <attribute name="name"/>
                  <choice>
                    <attribute name="url"/>
                    <ref name="anyElement"/>
                  </choice>
                </element>
              </zeroOrMore>
            </interleave>
          </element>
        </zeroOrMore>
        <zeroOrMore>
          <element name="stop-task">
            <element name="name">
              <data type="string"/>
            </element>
          </element>
        </zeroOrMore>
      </interleave>
    </element>
  </define>
  <define name="anyElement">
    <element>
      <anyName/>
      <zeroOrMore>
        <choice>
          <attribute>
            <anyName/>
          </attribute>
          <text/>
          <ref name="anyElement"/>
        </choice>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

You start a task by invoking the Scheduler processor with the `start-task` element. The following table describes the configuration elements.

name	The name of the task
------	----------------------

start-time	Either <code>now</code> or a date in Java's <code>DateFormat</code> format.
interval	Interval between tasks in milliseconds. If set to 0, the task runs only once.
synchronized	If set to <code>true</code> , the task won't run if the previous iteration is still running. Defaults to <code>false</code> .
processor-name	The qualified name of the processor that is executed when the task runs.
input	Inputs of the processor. The input can either be located in a separate file or inline.

In most cases, the task is described in an XPL pipeline. The following example shows how to use the `PipelineProcessor` to launch a periodic task.

```
<p: processor name="oxf:scheduler" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <config>
      <start-task>
        <name>myTask</name>
        <start-time>now</start-time>
        <interval>10000</interval>
        <processor-name>oxf:pipeline</processor-name>
        <input name="config" url="oxf:/mytask.xpl"/>
      </start-task>
    </config>
  </p: input>
</p: processor>
```

You can stop a task with the `stop-task` element and the name of the task.

```
<p: processor name="oxf:scheduler" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <stop-task>
      <name>myTask</name>
    </stop-task>
  </p: input>
</p: processor>
```

4.24. Other Processors

4.24.1. Resource Server

The Resource Server serves resources such as images, CSS stylesheet or other static files. Resources are sent to the HTTP response untouched, and the HTTP cache control headers are set.

The `config` input contains a single `url` element containing an absolute URL. The URL can be any URL supported by your platform, in particular URLs with the following protocols:

- `file`
- `http`
- `oxf` (to access OPS resources)

The Resource Server supports the deprecated use of a `config` input containing a single `path` element representing the absolute Resource Manager path of the file to serve. Since the `url` element also allows to access to OPS resources, it is recommended to use it instead of `path`.

The `mime-types` input contains a list of patterns and [MIME Media Types](#). This mapping list determines which `content-type` header to send to the browser. The patterns are case-insensitive. OPS is bundled with a default mapping file under the URL `oxf:/oxf/mime-types.xml`. You can create your own mapping to suit your needs. The RelaxNG schema is provided below.

```
<element name="mime-types" datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <oneOrMore>
    <element name="mime-type">
      <element name="name">
        <data type="string"/>
      </element>
      <oneOrMore>
        <element name="pattern">
          <data type="string"/>
        </element>
      </oneOrMore>
    </element>
  </oneOrMore>
</element>
```

The example below shows the Resource Server configured to send a PNG image file with the appropriate MIME type.

```
<p: processor name="oxf:resource-server" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
```

```

<p:input name="mime-types">
  <mime-types>
    <mime-type>
      <name>image/png</name>
      <pattern>*.png</pattern>
    </mime-type>
  </mime-types>
</p:input>
<p:input name="config">
  <url>oxf:/images/logo.png</url>
</p:input>
</p:processor>

```

4.24.2. Identity Processor

The Identity processor is one of the simplest processors of OPS: it simply copies the content of its data input to its data output. While at first this doesn't seem like a very useful feature, it actually can be very convenient, for example in the following scenarios.

XML documents can be embedded within pipeline inputs. When you want the same document to be passed to the input of multiple processors in a pipeline, instead of duplicating it and embedding it in every input, you can use the Identity processor: embed the document in the data input of the Identity processor, assign an id to the output of the Identity processor, and reference that id from other inputs you want to feed with that document.

```

<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <!-- Embed employee data -->
  <p:processor name="oxf:identity">
    <p:input name="data">
      <employees>
        <employee>
          <first-name>Joe</first-name>
          <last-name>Dalton</last-name>
        </employee>
        <employee>
          <first-name>Averell</first-name>
          <last-name>Dalton</last-name>
        </employee>
      </employees>
    </p:input>
    <p:output name="data" id="employees"/>
  </p:processor>
  <!-- Apply a first transformation on the employee data -->
  <p:processor name="oxf:xslt">
    <p:input name="config" href="oxf:/transform-1.xsl"/>
    <p:input name="data" href="#employees"/>
    <p:output name="data" id="result-1"/>
  </p:processor>
  <!-- Apply a second transformation on the same employee data -->
  <p:processor name="oxf:xslt">
    <p:input name="config" href="oxf:/transform-2.xsl"/>
    <p:input name="data" href="#employees"/>
    <p:output name="data" id="result-2"/>
  </p:processor>
  <!-- ... -->
</p:config>

```

Pipeline inputs support aggregation with the `aggregate()` function, as well as the XPointer syntax. The Identity processor can then be used to aggregate or otherwise modify existing documents in a pipeline, for example before sending a result to a pipeline output.

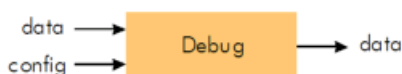
```

<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:param type="output" name="data"/>
  <!-- ... -->
  <!-- Assuming the XSLT transformations in the previous example, transform the two results and send them to the data output -->
  <p:processor name="oxf:identity">
    <p:input name="data" href="aggregate('result', #result-1#xpointer(/**/*), , #result-2#xpointer(/**/*[position() *10]))"/>
    <p:output name="data" ref="data"/>
  </p:processor>
</p:config>

```

4.24.3. Debug Processor

The Debug processor logs XML documents using the `Log4j` library. It has 2 inputs: data contains the XML document to log, and config has a config root element that contains a message typically be used to describe the XML document. The data output document is the exactly the same as the data input document. Consequently the debug processor can be easily inserted in a pipeline to log the XML data flow at a given point.



For instance, the processor can be called with:

```
<p:processor name="oxf:debug" xml:ns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>Employee</config>
  </p:input>
  <p:input name="data">
    <employee>
      <firstname>John</firstname>
      <lastname>Smith</lastname>
    </employee>
  </p:input>
  <p:output name="data" id="emp"/>
</p:processor>
```

This will generate the message:

```
Employee:
    <employee>
      <firstname>John</firstname>
      <lastname>Smith</lastname>
    </employee>
```

Using `debug` attributes in pipelines is a shortcut for actually inserting the Debug processor in the pipeline: the Pipeline processor will insert a Debug processor when encountering `debug` attributes. By changing the `processors.xml` you can map the `oxf:debug` processor to your own "Debug processor" and this processor will be called when you use `debug` attributes in pipelines. If you decide to implement your own debug processor, note that it must have the same interface as the default Debug processor that comes with OPS.

4.24.4. Redirect Processor

The Redirect Processor allows redirecting or forwarding the execution to a new URL:

- **Client-side** the browser is redirected to another URL. Typically, for a Servlet environment, the `sendRedirect()` method is called on the HTTP response object.

If the redirection URL is an absolute path (e.g. `/hello`) then it is interpreted as a path relative to the OPS Servlet context (i.e. if your OPS WAR file is under `/ops`, then the resulting path is `/ops/hello`). For other purposes, including redirecting to a path within the same Servlet container but outside the OPS WAR context, you have to use an absolute URL complete with scheme and host name.

- **Server-side**: a server-side forward is executed. Typically, for a Servlet environment, the `forward` method is called on a Servlet request dispatcher.

The processor's data input must conform to the following Relax NG schema:

```
<element name="redirect-url">
  <interleave>
    <optional>
      <element name="server-side">
        <choice>
          <value>true</value>
          <value>false</value>
        </choice>
      </element>
    </optional>
    <element name="path-info">
      <text/>
    </element>
    <optional>
      <element name="parameters">
        <zeroOrMore>
          <element name="parameter">
            <element name="name">
              <text/>
            </element>
            <oneOrMore>
              <element name="value">
                <text/>
              </element>
            </oneOrMore>
          </element>
        </zeroOrMore>
      </element>
    </optional>
  </interleave>
</element>
```

The optional boolean `server-side` element determines whether a server-side forward is performed. The default is `false`.

This example creates a processor that redirects the browser to `/login?user=jsmith`:

```
<p: processor name="oxf: redirect" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="data">
    <redirect-url>
      <path-info>/login</path-info>
      <parameters>
        <parameter>
          <name>user</name>
          <value>jsmith</value>
        </parameter>
      </parameters>
    </redirect-url>
  </p: input>
</p: processor>
```

Note

It is recommended, whenever possible, to use the [Page Flow Controller](#) to perform page redirections within a OPS application. The Page Flow Controller provides a much higher-level abstraction of the notion of redirection than the Redirect processor.

5. API Reference

5.1. Processor API

5.1.1. Scope

This section documents the OPS Processor API. This is a Java API that you can use to write custom processors. You can then use those custom processors in your OPS applications, just like the standard processors bundled with OPS.

5.1.2. Why Write Custom Processors?

In general, OPS processors encapsulate logic to perform generic tasks such as executing an XSLT transformation, calling a web service or accessing a database using SQL. With those processors, the developer describes the specifics of a task at a high level in a declarative way.

However, there are cases where:

- no existing processor exactly provides the functionality to be performed
- or, it is more suitable to write Java code to get the job done rather than using an existing processor

In those cases, it makes sense for the developer to write his own processor in Java. This section goes through the essential APIs used to write processors in Java.

Note

You can compile your processor yourself, or you can use the convenient [Java processor](#) which automatically compiles Java code on the fly.

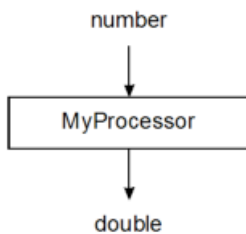
5.1.3. Prerequisites

Writing OPS processors is expected to be done by Java developers who are comfortable with the Java language as well as compiling and deploying onto servlet containers or J2EE application servers. In addition, we assume that the developer is comfortable with either:

- The [SAX API](#)
- The [W3C DOM](#) or [DOM4J](#) APIs.

5.1.4. Processors With Outputs

We consider a very simple processor with an input `number` and an output `double`. The processor computes the double of the number it gets as an input. For instance, if the input is `<number>21</number>`, the output will be `<number>42</number>`.



```
import org.orbeon.oxf.pipeline.api.PipelineContext;
import org.orbeon.oxf.processor.SimpleProcessor;
import org.orbeon.oxf.processor.ProcessorInputOutputInfo;
import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.AttributesImpl;
import org.dom4j.Document;

public class MyProcessor extends SimpleProcessor {

    public MyProcessor() {
        addInputInfo(new ProcessorInputOutputInfo("number"));
        addOutputInfo(new ProcessorInputOutputInfo("double"));
    }

    public void generateDouble(PipelineContext context,
                              ContentHandler contentHandler)
        throws SAXException {

        // Get number from input using DOM4J
        Document numberDocument = readInputAsDOM4J(context, "number");
        String numberString = (String)
            numberDocument.selectObject("string(/number)");
        int number = Integer.parseInt(numberString);
```

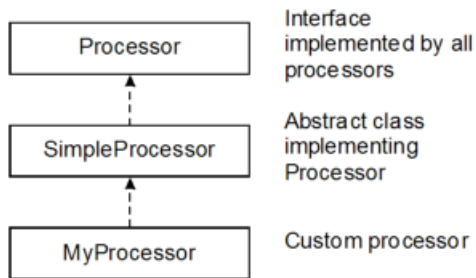
```

String doubleString = Integer.toString(number * 2);

// Generate output document with SAX
contentHandler.startDocument();
contentHandler.startElement("", "number", "number",
    new AttributesImpl());
contentHandler.characters(doubleString.toCharArray(), 0,
    doubleString.length());
contentHandler.endElement("", "number", "number");
contentHandler.endDocument();
}
}

```

All the processors must implement the `Processor` interface (in the package `org.orbeon.oxf.pipeline.processors`). `SimpleProcessor` is an abstract class that implements all the methods of `Processor` and that can be used as a base class to create a custom processor (`MyProcessor.java` in the figure below).



The processor must declare its mandatory static inputs and outputs. This is done in the default constructor by calling the `addInputInfo` and `addOutputInfo` methods and passing an object of type `ProcessorInputOutputInfo`. For instance:

```

public MyProcessor() {
    addInputInfo(new ProcessorInputOutputInfo("number"));
    addOutputInfo(new ProcessorInputOutputInfo("double"));
}

```

In addition to the name of the input/output, one can pass an optional schema URI declared in the [OPS properties](#). If a schema URI is specified, the corresponding input or output can be [validated](#).

Note

Note that the processor may have optional inputs and outputs, and/or read dynamic inputs and generate dynamic outputs, in which case it doesn't need to declare such inputs with `addInputInfo` and `addOutputInfo`.

For each declared output, the class must declare a corresponding `generate` method. For instance, in the example, we have an output named `double`. The document for this output is produced by the method `generateDouble`. `generate` methods must have two arguments:

- A `PipelineContext`. This context needs to be passed to other methods that need one, typically to read inputs (more on this later).
- A `ContentHandler`. This is a [SAX content handler](#) that receives the document produced by the `generate` method.

If the output depends on the inputs, one will need to read those inputs. There are 3 different APIs to read an input:

- One can get the [W3C DOM](#) representation of the input document by calling the `readInputAsDOM(context, name)` method.
- One can get the [DOM4J](#) representation of the input document by calling the `readInputAsDOM4J(context, name)` method.
- One can provide a custom [SAX content handler](#) to the method `readInputAsSAX(context, name, contentHandler)`.

Depending on what the `generate` method needs to do with the input document, one API might be more appropriate than the others.

In our example, we want to get the value inside the `<number>` element. We decided to go with the `DOM4J` API, calling the `numberDocument.selectObject("string(/number)")` on the `DOM4J` document.

The output document can alternatively be generated by:

- Directly calling methods of the content handler received by the `generate` method. This is what we do in the example detailed in this section. Here is the code generating the output document:

```

contentHandler.startDocument();
contentHandler.startElement("", "number", "number",
    new AttributesImpl());
contentHandler.characters(doubleString.toCharArray(), 0,

```

```

        doubleString.length());
contentHandler.endElement("", "number", "number");
contentHandler.endDocument();

```

- Create a DOM4J document and have it sent to the content handler using a `LocationSAXWriter` (in package `org.orbeon.oxf.xml.dom4j`):

```

Document doc = ...;
LocationSAXWriter saxWriter = new LocationSAXWriter();
saxWriter.setContentHandler(contentHandler);
saxWriter.write(doc);

```

Note

Using the `LocationSAXWriter` provided with OPS is the preferred way to write a DOM4J document to a SAX content handler. The standard JAXP API (calling `transform` with a `org.dom4j.io.DocumentSource`) can also be used, but if it is used, the location information stored in the DOM4J document will be lost.

- Create a W3C document and send it to the content handler using the standard JAXP API:

```

Document doc = ...;
Transformer identity = TransformerUtils.getIdentityTransformer();
transformer.transform(new DOMSource(doc), new SAXResult(contentHandler));

```

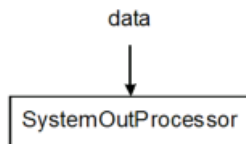
Note

`TransformerUtils` is a OPS class (in package `org.orbeon.oxf.xml`). It will create and cache the appropriate transformer factory. The developer is of course free to create its own factory and transformer calling directly the JAXP API.

5.1.5. Processors With No Output

Implementing a processor with no output is very similar to implementing a processor with outputs (see above). The only difference is that you need to implement the `start()` method, instead of the `generate()` methods.

The processor below reads its data input and writes the content of the XML document to the standard output stream.



```

package org.orbeon.oxf;

import org.dom4j.Document;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.XMLWriter;
import org.orbeon.oxf.common.OXFException;
import org.orbeon.oxf.processor.ProcessorInputOutputInfo;
import org.orbeon.oxf.processor.SimpleProcessor;
import org.orbeon.oxf.pipeline.api.PipelineContext;

import java.io.IOException;
import java.io.StringWriter;

public class SystemOutProcessor extends SimpleProcessor {

    public SystemOutProcessor() {
        addInputInfo(new ProcessorInputOutputInfo("data"));
    }

    public void start(PipelineContext context) {
        try {
            Document dataDocument = readInputAsDOM4J(context, "data");
            OutputFormat format = OutputFormat.createPrettyPrint();
            format.setIndentSize(4);
            StringWriter writer = new StringWriter();
            XMLWriter xmlWriter = new XMLWriter(writer, format);
            xmlWriter.write(dataDocument);
            xmlWriter.close();
            System.out.println(writer.toString());
        } catch (IOException e) {
            throw new OXFException(e);
        }
    }
}

```

The `PipelineContext` object is used to store information that must be kept for the entire execution of the current XPL program. This information is:

- Reset everytime the XPL program is run
- Separate for multiple concurrent executions of an XPL program.
- Shared among all the processors run during the XPL program's execution, including multiple instances of a given processor.

Use the following methods of `PipelineContext` to store XML program state:

```
/**
 * Set an attribute in the context.
 *
 * @param key the attribute key
 * @param o the attribute value to associate with the key
 */
public synchronized void setAttribute(Object key, Object o);
```

```
/**
 * Get an attribute in the context.
 *
 * @param key the attribute key
 * @return the attribute value, null if there is no attribute with the given key
 */
public Object getAttribute(Object key);
```

You can register a listener on the `PipelineContext` object to perform clean-up upon the termination of the XML program, using the following API:

```
/**
 * Add a new listener to the context.
 *
 * @param listener
 */
public synchronized void addContextListener(ContextListener listener);
```

```
/**
 * ContextListener interface to listen on PipelineContext events.
 */
public interface ContextListener {
    /**
     * Called when the context is destroyed.
     *
     * @param success true if the pipeline execution was successful, false otherwise
     */
    public void contextDestroyed(boolean success);
}
```

```
/**
 * ContextListener adapter class to facilitate implementations of the ContextListener
 * interface.
 */
public static class ContextListenerAdapter implements ContextListener {
    public void contextDestroyed(boolean success) {
    }
}
```

You can register a listener as follows:

```
pipelineContext.addContextListener(new ContextListenerAdapter() {
    public void contextDestroyed(boolean success) {
        // Implement your clean-up code here
    }
});
```

Examples of clean-up operations include:

- Performing commits or rollbacks on external resources
- Freeing-up external resources allocated for the execution of the XPL program only

Processors with multiple outputs often have to perform some task when the first output is read, store the result of the task, and then make it available to the other outputs when they are read. This information is:

- Reset everytime the XPL program is run.
- Separate for every processor instance.
- Shared between calls of the `start()`, `ProcessorOutput.readImpl()` and `generateXxx()` of a given processor instance, during a given XPL program execution..

The `PipelineContext` methods are not sufficient for this purpose. In order to store state information tied to the current execution of the current processor, and shared across the current processor's initialization as well as outputs reads, use the following methods:

```
/**
 * This method is used by processor implementations to store state
 * information tied to the current execution of the current processor,
 * across processor initialization as well as reads of all the
 * processor's outputs.
 *
 * This method should be called from the reset() method.
 *
 * @param context current PipelineContext object
 * @param state    user-defined object containing state information
 */
protected void setState(PipelineContext context, Object state);
```

```
/**
 * This method is used to retrieve the state information set with setState().
 *
 * This method may be called from start() and ProcessorOutput.readImpl().
 *
 * @param context current PipelineContext object
 * @return         state object set by the caller of setState()
 */
protected Object getState(PipelineContext context);
```

You initialize the processor state in the `reset()` method, as follows:

```
public void reset(PipelineContext context) {
    setState(context, new State());
}
```

Where you define class `State` as you wish, for example:

```
private static class State {
    public Object myStuff;
    ...
}
```

You can then obtain your execution state by calling the `getState()` method:

```
State state = (State) getState(context);
```

You call `getState()` from the `start()`, `ProcessorOutput.readImpl()` or `generateXxx()`.

5.1.7. Using custom processors from XPL

In order to use a custom processor compiled and deployed within OPS (as one or more class files or JAR files), its main class (the one that implements the `Processor` interface) must be mapped to an XML qualified name. You do this mapping in a file called `custom-processors.xml` under the `config` directory. This is an example illustrating the format of the file:

```
<processors>
  <processor name="my:processor">
    <class name="com.company.my.MyOtherProcessor"/>
  </processor>
  <processor name="my:system-out">
    <class name="com.company.my.SystemOutProcessor"/>
  </processor>
</processors>
```

You choose your own prefix name and namespace URI for that prefix, in this example `my` and `http://my.company.com/ops/processors` respectively. There is no strict format for the namespace URI, but it should identify your company or organization, here with `my.company.com`.

Do the mapping on the `processors` element as usual in XML with:

```
xml ns:my="http://my.company.com/ops/processors"
```

You use processors in XPL using the same qualified names used in `custom-processors.xml`:

```
<p:processor name="my:system-out" xml ns:p="http://www.orbeon.com/oxf/pipeline">...</p:processor>
```

It is important to declare the namespace mapping in `custom-processors.xml` as well as in the XPL programs that use those processors.

There is no constraint to use a single namespace: you may declare multiple namespaces to categorize processors. Using one namespace per company or organization is however the recommended option for convenience.

5.2. Pipeline Engine API

5.2.1. Introduction

The pipeline engine API allows embedding the execution of pipelines in your Java applications, whether server-side or client-side.

5.2.2. API

The reference example using those APIs is the command-line application found under `OPS.java`. The steps required to execute a pipeline are detailed below. Please also refer to the source code of `OPS.java`.

Please refer to the source code for more details.

Pipelines usually access resources (or files) through an abstraction layer composed of one or more [resource managers](#). The code below initializes a Priority Resource Manager, which attempts to load resources first from the filesystem, then from the classloader. The reason for using the classloader is that the Orbeon JAR file contain bundled configuration resources that must be accessible.

In this particular case, if the `-r` command-line argument is present, a sandbox directory path is passed to the Filesystem resource manager upon instantiation.

```
Map props = new HashMap();
props.put("oxf.resources.factory", "org.orbeon.oxf.resources.PriorityResourceManagerFactory");
if (resourceManagerSandbox != null) {
    // Use a sandbox
    props.put("oxf.resourceManager.filesystem.sandbox-directory", resourceManagerSandbox);
}
props.put("oxf.resources.priority.1", "org.orbeon.oxf.resources.FilesystemResourceManagerFactory");
props.put("oxf.resources.priority.2", "org.orbeon.oxf.resources.ClassLoaderResourceManagerFactory");
if (logger.isInfoEnabled())
    logger.info("Initializing Resource Manager with: " + props);
ResourceManagerWrapper.init(props);
```

This initializes the OPS properties with the default properties file bundled in the Orbeon JAR file.

```
OXFProperties.init(OXFProperties.DEFAULT_PROPERTIES_URI);
```

This initializes a logger.

```
LoggerFactory.initLogger();
```

This step builds a `ProcessorDefinition` object containing the name of the processor to run (here the `oxf:pipeline` processor), as well as the URL to bind to the config input of that processor. The mapping of processor names to classes is done in `processors.xml`, a resource bundled in the Orbeon JAR file.

```
if (otherArgs != null && otherArgs.length == 1) {
    // Assume the pipeline processor and a config input
    processorDefinition = new ProcessorDefinition();
    processorDefinition.setName(new QName("pipeline", XMLConstants.OXF_PROCESSORS_NAMESPACE));

    String configURL;
    if (!NetUtils.urlHasProtocol(otherArgs[0])) {
        // URL is considered relative to current directory
        try {
            // Create absolute URL, and switch to the oxf: protocol
            String fileURL = new URL(new File(".").toURL(), otherArgs[0]).toExternalForm();
            configURL = "oxf:" + fileURL.substring(fileURL.indexOf(':') + 1);
        } catch (MalformedURLException e) {
            throw new OXFException(e);
        }
    } else {
        configURL = otherArgs[0];
    }

    processorDefinition.addInput("config", configURL);
} else {
    throw new OXFException("No main processor definition found.");
}
```

The `PipelineContext` represents a context object passed to all the processors running in a given pipeline session. In general, you just need to create an instance.

```
PipelineContext pipelineContext = new PipelineContext();

// Some processors may require a JNDI context. In general, this is not required.
Context jndiContext;
try {
    jndiContext = new InitialContext();
} catch (NamingException e) {
    throw new OXFException(e);
}
pipelineContext.setAttribute(PipelineContext.JNDI_CONTEXT, jndiContext);
```

```
pipelineBuilder.addStep(instanceOf(ExecutionPipelineProcessor.class), new CommandExecutionContext(),  
pipelineContext);
```

If an exception is caught, information about the error is displayed.

```
LocationData locationData = ValidationException.getRootLocationData(e);  
Throwable throwable = OXFException.getRootThrowable(e);  
String message = locationData == null  
    ? "Exception with no location data"  
    : "Exception at " + locationData.toString();  
logger.error(message, throwable);
```

6. Integration

6.1. Authentication

6.1.1. Introduction

Most Web applications require some sort of authentication, either just to access an "administrative" section, or for the whole application. OPS Web applications use the standard authentication mechanism provided by your J2EE application server. This section provides an overview of the standard J2EE authentication mechanism.

6.1.2. Restricting Access using web.xml

Access control is provided by adding three sections to the `web.xml` file:

- In `<security-constraint>` you define which role (with `<role-name>`) is required to access which part of the application (with `<url-pattern>`).

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Administration</web-resource-name>
    <url-pattern>/admin</url-pattern>
    <url-pattern>/users</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
  </auth-constraint>
</security-constraint>
```

- In `<login-config>` you define how the user will authenticate himself. In other words, what method is used to get the user name and password. This can be done either with a form in an HTML page or with standard HTTP authentication. The names of those methods are: `FORM` and `BASIC`. In the example below, the form mechanism is demonstrated. `<form-login-page>` must point to a page with an HTML form where:

- The form action is set to `j_security_check`.
- The name of the field used to get the username is `j_username`.
- The name of the field used to get the password is `j_password`.

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login</form-login-page>
    <form-error-page>/login-error</form-error-page>
  </form-login-config>
</login-config>
```

- In `<login-config>` the security roles used in `<security-constraint>` section are declared.

```
<security-role>
  <role-name>administrator</role-name>
</security-role>
```

6.1.3. Mapping Roles to Users

In the `web.xml` file, the example declared that to access the page `/admin` the user needs to have the `administrator` role. But how do you declare users and how are those users mapped to roles? This is application server dependent, so you won't find an exact answer to this question in the OPS User Guide and you should refer to your application server documentation.

Usually the process is straightforward. For example, with Tomcat using the memory realm, you can declare the users and their role in `conf/tomcat-users.xml`:

```
<tomcat-users>
  <user name="root" password="olleh" roles="administrator"/>
  <user name="jdoh" password="olleh" roles="administrator"/>
</tomcat-users>
```

For more information on how to setup users and assign roles to users, see your application server documentation. Links are provided below for Tomcat and WebLogic.

- [Tomcat 4.0](#)
- [Tomcat 4.1](#)
- [WebLogic 6.1](#)
- [WebLogic 7.0](#)

6.1.4. Accessing Security Information From the Application

The Request Security processor extracts information about the currently logged user from the client request. Its configuration contains a list of roles the application developer is interested in. Only those roles will be listed in the processor's output if the role is present. For instance, the output of the Request Security processor could be:

```
<request-security>
  <auth-type>FORM</auth-type>
  <secure>true</secure>
  <remote-user>scott</remote-user>
  <role>user</role>
</request-security>
```

The `auth-type` element contains either BASIC, FORM, CLIENT_CERT, or DIGEST. The `secure` element is true if the request was made using a secure channel, such as HTTPS. See the [Servlet API](#) for more information.

```
<p:processor name="oxf:request-security" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config>
      <role>user</role>
      <role>admin</role>
    </config>
  </p:input>
  <p:output name="data" id="request-security"/>
</p:processor>
```

6.1.5. Logout

In order to log the current user out, the Session Invalidator processor must be used:

```
<p:processor name="oxf:session-invalidator" xmlns:p="http://www.orbeon.com/oxf/pipeline"/>
```

The Session Invalidator processor does not take any configuration or other inputs and outputs. It must be included in a pipeline or branch of pipeline executed when the action of logging out the user is requested.

6.2. Command Line Applications

6.2.1. Introduction

You can use OPS to build standalone command-line applications running pipelines. Use cases include: creating a hardcopy of a web site, importing or exporting XML data to and from a relational database, testing pipelines, automating repetitive operations on XML files (selection, transformation, aggregation, etc.). OPS becomes your "XML toolbox".

6.2.2. Command Line Interface

OPS ships with a command line interface (CLI). The easiest way to run it is to use the executable `cli-ops.jar` file and pass it a pipeline file:

```
java -jar cli-ops.jar pipeline.xml
```

In this case, a pipeline can use relative URLs, or, if using absolute URLs, can use either the `file:` protocol to access files or, equivalently, the `oxf:` protocol.

Sometimes, it is better to group files under a resources *sandbox* addressed by the `oxf:` protocol. This is the standard way of accessing resources (files) within web applications, but it is also useful with command-line applications. In such cases, specify a resource manager root directory with the `-r` option, for example:

```
java -jar cli-ops.jar -r . oxf:/pipeline.xml
```

or:

```
java -jar cli-ops.jar -r C:/my-example oxf:/pipeline.xml
```

When specifying a resource manager root directory, it is mandatory to use a protocol and an absolute path within the sandbox, as shown above.

Note

OPS's jar file (`cli-ops.jar`) is executable via the `java -jar` mechanism. However, it depends on a number of libraries that must be in the 'lib' directory beneath the directory containing `cli-ops.jar`. It is recommended to expand the WAR file, and refer to the `cli-ops.jar` file under the WAR file's *WEB-INF* directory.

6.2.3. Examples

OPS comes with simple examples of command-line applications under the `src/examples/cli` directory. To run those examples, from a command prompt, enter either one of the subdirectories and run the `java` command following the explanations above, for example, from the `simple` directory:

```
java -jar /path/to/orbeon-war/WEB-INF/cli-ops.jar stdout.xml
```

or, from the `transform` directory:

```
java -jar /path/to/orbeon-war/WEB-INF/cli-ops.jar transform.xml
```

Notice that those two example pipelines use serializers, for example:

```
<p: config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: processor name="oxf:xslt">
    <p: input name="data" href="foo.xml"/>
    <p: input name="config" href="foo.xml"/>
    <p: output name="data" id="result"/>
  </p: processor>
  <p: processor name="oxf:xml-serializer">
    <p: input name="config">
      <config/>
    </p: input>
    <p: input name="data" href="#result"/>
  </p: processor>
</p: config>
```

It is important to note that, when running from within a servlet or portlet environment, serializers send their data to the web client, such as a web browser. When running from a command-line environment, serializers output data to the standard output.

6.3. Web Services

6.3.1. Overview

Your OPS applications can consume and expose Web services.

6.3.2. Consuming Web Services

Applications built on OPS call Web services using the [delegation processor](#). For example, this is how you would call the famous highway traffic Web service to get the conditions on the 101 highway in California from an XPL:

```
<p: processor name="oxf:delegation" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="interface">
    <config>
      <service id="ca-traffic" type="web-service" endpoint="http://services.xmethods.net/soap/servlet/rpcrouter">
        <operation nsuri="urn:xmethods-CATraffic" name="getTraffic"/>
      </service>
    </config>
  </p: input>
  <p: input name="call">
    <delegation:execute service="ca-traffic" operation="getTraffic" xmlns:delegation="http://orbeon.org/oxf/xml/delegation">
      <hwynums>101</hwynums>
    </delegation:execute>
  </p: input>
  <p: output name="data" id="traffic"/>
</p: processor>
```

The output of that call would be:

```
<return xsi:type="xsd:string">
  reported as of Wednesday, July 2, 2003 at 16:18 . Slow for the Cone Zone US 101 [LOS ANGELES & VENTURA CO.'S] NO
  TRAFFIC RESTRICTIONS ARE REPORTED FOR THIS AREA ...
</return>
```

6.3.3. Exposing Web Services

To expose a Web service, follow these steps:

1. Declare the Web service in the `page-flow.xml`, just like a regular page with no view. The Web services will be implemented in an XPL file. The OPS example portal features a Web service that sends instant messages to Yahoo! users. It is declared in the `page-flow.xml` as follows:

```
<page path-info="/examples/im" model="oxf:/im/yim-web-service.xpl"/>
```

2. To get the SOAP envelope sent to the Web service, use the [Request generator](#) and include the path `/request/body`, specifying the `xs:anyURI` type for the result. The resulting URI can then be parsed with the [URL generator](#):

```
<!-- Extract request body as a URI -->
<p: processor name="oxf:request" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p: input name="config">
    <config stream-type="xs:anyURI">
      <include>/request/body</include>
    </config>
  </p: input>
  <p: output name="data" id="request"/>
</p: processor>
<!-- Dereference URI -->
<p: processor name="oxf:url-generator" xmlns:p="http://www.orbeon.com/oxf/pipeline">
```

```

    <p:input name="config" href="aggregate('config', aggregate('url', #request#xpath(//request/body)))/">
  >
  <p:output name="data" id="file"/>
</p:processor>

```

You will find the SOAP envelope sent by the client of your Web service as the resulting file. For example:

```

<soapenv:Envelope soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>...</soapenv:Body>
</soapenv:Envelope>

```

3. Generate a SOAP envelope that responds to the client's request and send it to the client with the [XML serializer](#), as follows:

```

<p:processor name="oxf:xml-serializer" xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:input name="config">
    <config/>
  </p:input>
  <p:input name="data" href="#response"/>
</p:processor>

```

6.4. Packaging and Deployment

6.4.1. Scope

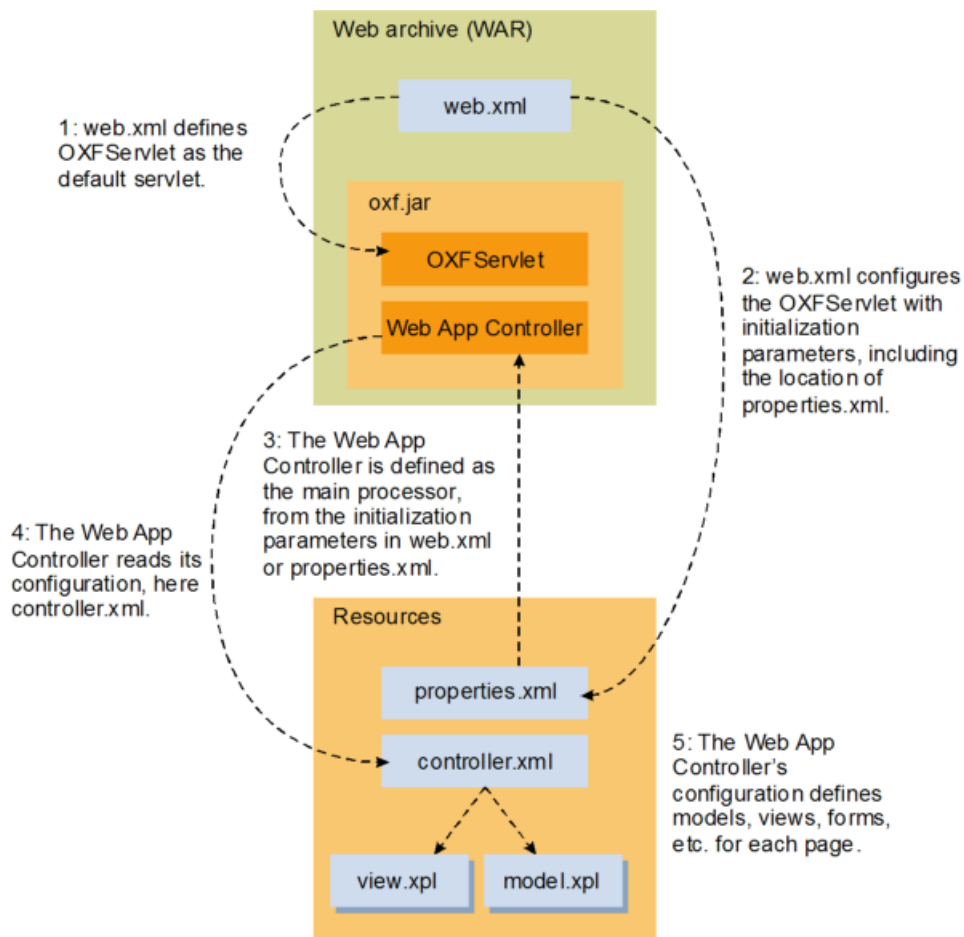
This section explains the structure of the standard WAR distributed with OPS and how this WAR integrates with the application server (or Servlet container). This information is useful if you need to repackage Presentation Server. For instance, if you want to build an EAR file, or if you need to deploy more complex OPS applications containing Servlets, Servlet Filters, Portlets, and Servlet context listeners.

6.4.2. WAR Structure

Files	Description
WEB-INF/lib/ops.jar	JAR file with all the OPS classes.
WEB-INF/lib/*.jar	All the other JAR files in the WEB-INF/lib directory are used either by the OPS core engine, or one of the Presentation Server processors.
WEB-INF/web.xml	The standard descriptor for this WAR file. It declares <code>OXFServlet</code> as the default Servlet and passes some basic configuration parameters to this Servlet.
WEB-INF/portlet.xml	The standard portlet descriptor for this WAR file is required if you use portlets. It typically declares instances of <code>OXFPortlet</code> . For more information, see Writing Portlets With OPS .
WEB-INF/weblogic.xml	An additional descriptor for WebLogic. This descriptor typically maps resource names to actual resources configured in the application server (e.g. for EJBs, users, JDBC data sources, etc).
WEB-INF/sun-web.xml	An additional descriptor for SunOne. This descriptor typically maps resource names to actual resources configured in the application server (e.g. for EJBs, users, JDBC data sources, etc).
WEB-INF/resources/*	Contains the resources for the OPS example application. Most of the files in this directory will be replaced when you build your own OPS web applications. In particular, this directory contains <code>properties.xml</code> , the main OPS configuration file.

6.4.3. OPS Initialization

The following figure illustrates the initialization of a simple OPS deployment in a J2EE application server:



The initialization follows this lifecycle:

1. The application server reads the `WEB-INF/web.xml` file, which:

- Declares a Servlet named `oxf` implemented by the class `org.orbeon.oxf.servlet.OPSServlet` (loaded from `lib/ops.jar`)
- Defines `oxf` as the default Servlet (i.e. the Servlet handling all the requests).

```
<web-app>
  <servlet>
    <servlet-name>oxf</servlet-name>
    <servlet-class>org.orbeon.oxf.servlet.OPSServlet</servlet-class>
    <!-- Initialization parameters here (see below) -->
  </servlet>
  <servlet-mapping>
    <servlet-name>oxf</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

2. The `web.xml` file configures the OXFServlet with a minimal set of parameters. Those parameters tell OXFServlet:

- What resource manager has to be used, and how this resource manager is configured. In the default WAR bundled in the OPS distribution, OPS loads resources from the `WEB-INF/resources` directory inside the WAR. If it can't find a resource in this directory, it will try to look for it inside `ops.jar`. Only static resources that are part of OPS are stored in `ops.jar` (as opposed to OPS applications). The [Resource Managers](#) section explains in detail how resource managers work.
- The location of `properties.xml`.
- Optionally, what main processor or context listener processors must be used.

3. OPS is configured through an XML file, `properties.xml`, stored with the resources. The exact name and path of this file is specified within `web.xml`. `properties.xml` may declare the main processor to be executed by OXFServlet, as well as optional inputs of this processor. Alternatively, the main processor can be declared directly within `web.xml`, which is the recommended approach. By default, the Page Flow Controller is used as the main processor. If the configuration is done in `web.xml`:

```
<!-- The main processor that OXFServlet must execute -->
<context-param>
  <param-name>oxf.main-processor.name</param-name>
  <param-value>{http://www.orbeon.com/oxf/processors}page-flow</param-value>
</context-param>
<!-- The Page Flow Controller configuration file -->
<context-param>
  <param-name>oxf.main-processor.input.controller</param-name>
```



```
<param-val ue>oxf: /page-fl ow. xml </param-val ue>
</context-param>
```

If the configuration is done in `properties.xml`:

```
<properties>
<!-- The main processor that OXFServlet must execute -->
<property as="xs: QName" name="oxf: mai n-processor. mai n" val ue="oxf: page-fl ow"/>
<!-- The Page Flow Controller configuration file -->
<property as="xs: anyURI " name="oxf. mai n-processor. i nput. control l er" val ue="oxf: /page-fl ow. xml "/>
</properties>
```

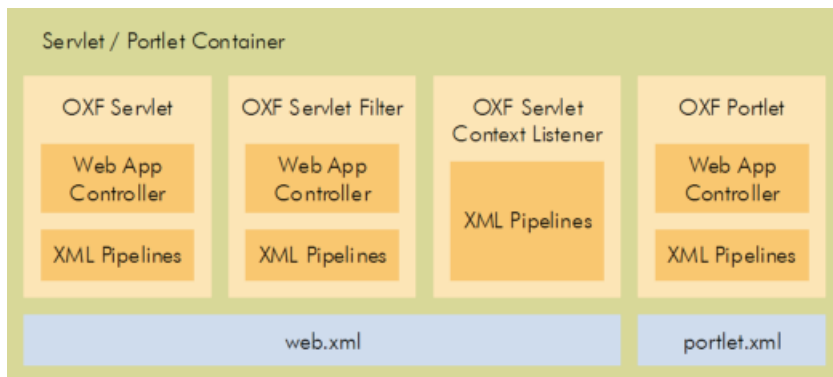
For more information about the properties file, see [OPS Properties](#).

4. The `oxf.main-processor.input.controller` property connects the controller input of the Page Flow Controller to the configuration file `oxf: /page-flow.xml`. The Page Flow Controller reads this input before it starts to operate.
5. The Page Flow Controller now handles client requests and dispatches them to other pipelines. For more information about the role of the controller, see the [Page Flow Controller reference](#).

6.4.4. Main Processor

In the same way that an old-fashioned program has a main function, Presentation Server has the concept of main processor. Within a web application, the main processor is the processor that is run each time a Servlet, Servlet filter or Portlet receives a client request. Within a command-line application, the main processor is simply the processor that runs when the application is run.

In the simplest web application deployment scenario, as shown in the example above, only one OPS Servlet needs to be configured. In more complex scenarios, it is possible to deploy multiple OPS Servlets, Servlet filters, and Portlets, as well as one OPS Servlet context listener, within the same Web or Portlet Application. The following figure illustrates this:



Additional non-OPS components can obviously be deployed within the same Web or Portlet Application.

These components can each have their own main processor. The main processor for such components is looked up in the following locations, in this order:

1. The component's initialization parameters in `web.xml`. For example, in the case of a Servlet:

```
<servlet>
<servlet-name>oxf</servlet-name>
<servlet-class>org. orbeon. oxf. servlet. OPSServlet</servlet-class>
<!-- The main processor that OXFServlet must execute -->
<init-param>
<param-name>oxf. mai n-processor. name</param-name>
<param-val ue>{http: //www. orbeon. com/oxf/processors}page-fl ow</param-val ue>
</init-param>
<!-- The Page Flow Controller configuration file -->
<init-param>
<param-name>oxf. mai n-processor. i nput. control l er</param-name>
<param-val ue>oxf: /page-fl ow. xml </param-val ue>
</init-param>
</servlet>
```

2. `properties.xml`, for example:

```
<properties>
<!-- The main processor that OXFServlet must execute -->
<property as="xs: QName" name="oxf. mai n-processor. name" val ue="oxf: page-fl ow"/>
<!-- The Page Flow Controller configuration file -->
<property as="xs: anyURI " name="oxf. mai n-processor. i nput. control l er" val ue="oxf: /page-fl ow. xml "/>
</properties>
```

3. The context parameters in `web.xml`

```

<!-- The main processor that OXFServlet must execute -->
<context-param>
  <param-name>oxf.main-processor.name</param-name>
  <param-value>{http://www.orbeon.com/oxf/processors}page-flow</param-value>
</context-param>
<!-- The Page Flow Controller configuration file -->
<context-param>
  <param-name>oxf.main-processor.input.controller</param-name>
  <param-value>oxf:/page-flow.xml</param-value>
</context-param>

```

It is recommended to configure each Web component individually in the component's initialization properties in `web.xml`, so that adding components with different configurations is facilitated. There are situations where several components need to share a configuration, but it is expected that such situations will be rare.

6.4.5. Error Processor

In case an error is encountered during the execution of the main processor, OPS tries to execute an error processor. The error processor is typically a pipeline that produces a page showing the exception that was encountered. For more information, please refer to the [Error Pipeline](#) documentation.

You can configure an error processor in the same way the main processor is configured. The error processor is looked up in the following locations, in this order:

1. The component's initialization parameters in `web.xml`. For example, in the case of a Servlet:

```

<servlet>
  <servlet-name>oxf</servlet-name>
  <servlet-class>org.orbeon.oxf.servlet.OPSServlet</servlet-class>
  <!-- The error processor that OXFServlet must execute -->
  <init-param>
    <param-name>oxf.error-processor.name</param-name>
    <param-value>{http://www.orbeon.com/oxf/processors}pipeline</param-value>
  </init-param>
  <!-- The pipeline to execute -->
  <init-param>
    <param-name>oxf.error-processor.input.config</param-name>
    <param-value>oxf:/config/error.xpl</param-value>
  </init-param>
</servlet>

```

2. `properties.xml`, for example:

```

<properties>
  <!-- The error processor that OXFServlet must execute -->
  <property as="xs:QName" name="oxf.error-processor.name" value="oxf:pipeline"/>
  <!-- The Page Flow Controller configuration file -->
  <property as="xs:anyURI" name="oxf.error-processor.input.config" value="oxf:/config/error.xpl"/>
</properties>

```

3. The context parameters in `web.xml`

```

<!-- The error processor that OXFServlet must execute -->
<context-param>
  <param-name>oxf.error-processor.name</param-name>
  <param-value>{http://www.orbeon.com/oxf/processors}pipeline</param-value>
</context-param>
<!-- The pipeline to execute -->
<context-param>
  <param-name>oxf.error-processor.input.config</param-name>
  <param-value>oxf:/config/error.xpl</param-value>
</context-param>

```

It is recommended to configure each Web or Portlet Application component individually in the component's initialization properties in `web.xml`, so that adding components with different configurations is facilitated. There are situations where several components need to share a configuration, but it is expected that such situations will be rare.

6.5. EJB Support

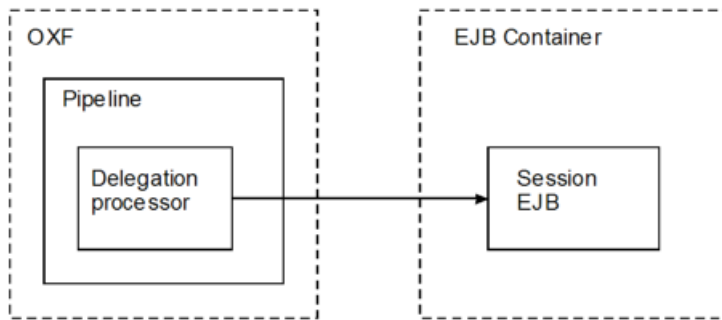
6.5.1. Classification

OPS provides EJB support in two different areas:

- Calling existing EJBs
- Encapsulating a processor in an EJB

6.5.2. Calling Existing EJBs

Regular EJBs (typically implementing business logic) can be called from a pipeline using the Delegation processor. For instance, this could be used in a Web application to call a credit card validation service available in an EJB session bean to validate a number entered by a user on a Web page. The Delegation processor can not only call EJB session beans, but also Web services and JavaBeans. Please see the [Delegation processor](#) documentation for more information about how to call existing EJBs.



6.5.3. Processor EJB Encapsulation

When code is encapsulated inside an EJB, the EJB container will provide a number of features to the developer almost hassle-free (i.e. with no coding involved), including:

- Load balancing: the application can be deployed on a cluster of servers and the application server will take care of load balancing.
- Transaction handling: one can define in a declarative way (XML in the ejb-jar.xml) the transactional behavior of a component. For example, when deciding which set of operations need to be executed in a single transaction. Then, the application server will automatically do whatever is necessary to handle those transactions properly.
- Code distribution: one can decide at deploy-time to deploy EJB A on a given server and EJB B on another remote server (perhaps because B is intensively accessing a database local to that remote server).

With Processor EJB Encapsulation, OPS processors can benefit from the same advantages provided to EJB components by the application server.

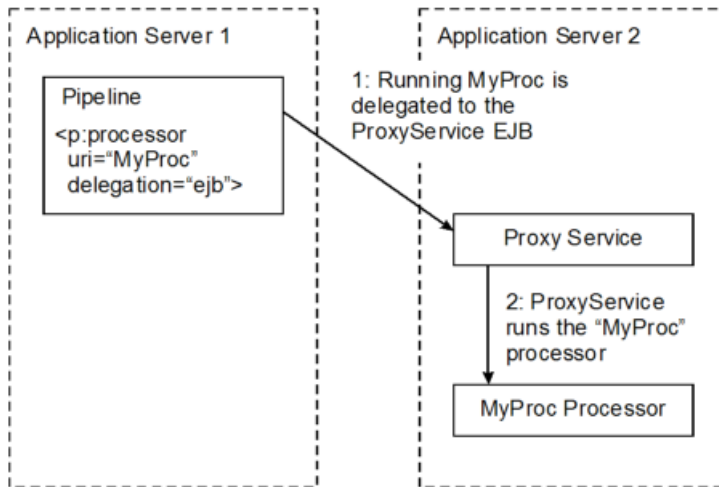
In the pipeline language, the attribute `delegation="ejb"` can be added to any `<p:processor>` element.

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline">
  <p:processor name="oxf:sql" delegation="ejb">
    <p:input name="config">...</p:input>
    <p:input name="data">...</p:input>
    <p:output name="data" id="result-set"/>
  </p:processor>
  <p:processor name="oxf:xslt">
    <p:input name="config">...</p:input>
    <p:input name="data" ref="result-set">...</p:input>
    <p:output name="data" id="result-set"/>
  </p:processor>
</p:config>
```

In the above example we first execute some SQL using the SQL processor and then transform the returned data via the XSLT processor. Since we have a `delegation="ejb"` attribute:

1. The pipeline processor will get the home interface stub by looking up `java:comp/env/ejb/oxf/proxy` from the JNDI tree.
2. From the home interface stub, the pipeline processor creates the Proxy Service stub and then calls the service.
3. Wherever it is deployed, the service creates and runs the processor.

The figure below illustrates a scenario where the Proxy Service EJB is deployed on a remote server.



To find the Proxy Service EJB, the pipeline processor must be able to lookup the name `java:comp/env/ejb/oxf/proxy`. Typically for Web applications, this pipeline will be called from the OXFServlet which is deployed in a WAR. In that case, the WAR's web.xml will have to declare an EJB reference:

```

<ejb-ref>
  <ejb-ref-name>ejb/oxf/proxy</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>org.orbeon.oxf.proxy.ProxyServiceHome</home>
  <remote>org.orbeon.oxf.proxy.ProxyService</remote>
</ejb-ref>
  
```

Depending on the application server, additional configuration will be needed. For instance, on WebLogic, the WAR's weblogic.xml descriptor might look like this:

```

<weblogic-web-app>
  <reference-descriptor>
    <ejb-reference-description>
      <ejb-ref-name>ejb/oxf/proxy</ejb-ref-name>
      <jndi-name>org/orbeon/oxf/ProxyService</jndi-name>
    </ejb-reference-description>
  </reference-descriptor>
</weblogic-web-app>
  
```