axiell

Programming XSLT stylesheets for Axiell Collections

Axiell ALM Netherlands BV

axiell

Contents

Introduction

Underneath the surface of the graphical user interface of your Axiell Collections application, records are handled by the software in XML format, which basically is a hierarchically structured text format. Normally, you won't encounter the XML itself, but you'll have to know about it if you want to start using XSLT to create inline reports, output formats or to edit presentation formats for your Axiell Internet Server web application, to name but a few. XSLT is a stylesheet language (itself in XML format) to "transform" an XML document to some other document; this may be an XML document with the same structure but with changes made to the data in it, or it can be a differently structured XML document, an HTML document or some other text file.
XML and XSLT are third-party programming technologies, so to properly learn all about them we recommend studying other sources than the document before you. In this manual though, you'll get a quick introduction followed by actual examples applicable to the Axiell software, enough to get you started properly.

1 An introduction to XML and XSLT

1.1 What is XML

XML (eXtentible Markup Language) provides a means of hierarchically structuring data in a text file. In contradiction to HTML (which is intended to lay out text or data for display in web browsers), it does not offer layout instructions. Other than a few PIs (Processing Instructions providing metadata about the document for the processor, enclosed in `<?  ?>`) at the start of the file, the only language it contains consists of tags, the name of which can be anything the maker of the XML document desired. Every separate piece of data must be enclosed by a start and end tag, formatted like `<tag>data</tag>`, together called an element or node, which may be spread over different lines. The following is an example of a simple yet complete XML document (not Axiell XML in any way though), although no title has been specified for the third book:

```
<?xml version="1.0" ?>
<!-- my comment -->
<booklist>
  <book isbn="901234567">
    <author>Hesse, Herman</author>
    <author>Claus, Hugo</author>
    <title>Siddharta</title>
  </book>
  <book>
    <author>Wolkers, Jan</author>
    <title>Terug naar Oegstgeest</title>
    <publisher>Summer &amp; Köning</publisher>
  </book>
  <book>
    <author>Austen, Jane</author>
  </book>
</booklist>
```

Every XML document has to start with: `<?xml version="1.0" ?>` or `<?xml version="2.0" ?>` to tell the processor which XML version is implemented in this document. Optionally, you could also mention here the Unicode representation in which this file has been saved, e.g.: `<?xml version="1.0" encoding="UTF-8"?>`

1.1.1 XML document requirements

There are some further rules to putting together an XML document.

1. Each XML document can only have one root tag. In the example above this is `booklist`.

2. Tags must have sensible names, so that others can easily understand the document, and for the sake of interoperability. Even if it contains Collections data, those names need not be the same as field names per se, but it does make it easier to process of course.

3. Every element must be closed. A start tag without an end tag corrupts an XML document. If there is no data between a start and end tag, this may be indicated by a single combined tag to open and close at once: `<tag/>`.
   Note that there's a difference between an empty tag, for instance `<title></title>` and no title tags at all, which is relevant to XSLT stylesheets processing an XML document.

4. Tags are nested. In the example you can see that the authors Hesse and Claus are nested within the first `book` tag, and that the `book` tags are nested within the root tag `booklist`. This nesting is crucial to keeping data together, like it is in Collections records.

5. The increasing indentation (whitespace) in front of nested tags, as shown in the example, is not strictly necessary, but it keeps the document readable.
   Visual Studio is handy for writing and editing XML docs, because it suggests end tags and adds coloring, but you can edit an XML doc in any text editor, like Notepad++, as long as you save the file in Unicode UTF-8 representation (if you want to use the XML file in Axiell software).

6. A tag may have attributes. An attribute is metadata included in a start tag, and it's purpose is to describe something about the data in the current element or all elements nested in it. It should be in the format `<tag attribute="value">`. In our example there is one attribute for the tag book: `<book isbn="901234567">`. This may not be a good example because ISBN is a field in a Collections record, and is not really considered metadata in there. But the maker of the document decides what is metadata and what isn't. The language of records could also be specified this way, for example: `<booklist language="en-US">`. Every XML start tag may have zero, one or more attributes; attributes should be separated by a space. And every attribute must have a unique name, specified by the maker of the document, and it cannot contain spaces. The double quotes around the value are mandatory, a value in between isn't. Note that double quotes come in different varieties, but should be the straight version, as follows: ", not “ or ” as created by MS Word for example.

7. A few characters have to be "escaped" (meaning: replaced) when used in the data itself, because they are reserved characters to the XML language. These are:

Character	Escape sequence to use in data
<	&lt;
>	&gt;
‘	&apos;
“	&quot;
&	&amp;

   In our example we see an illustration of this: `<publisher>Summer &amp; Köning</publisher>`. Note that other special characters in data, for instance á or € don't need to be escaped because this is a Unicode file.

8. XML tags and attributes are case-sensitive. So `<Author>` is not the same as `<author>`.

The easiest way of checking whether an XML document doesn't contain any errors is by double-clicking it in Windows Explorer. If the file opens normally in your internet browser, there are no XML syntax errors. However if there *are* syntax errors, then the file doesn't open and the browser displays an error message. So, your browser can validate an XML document.

1.1.2 How XML documents may be structured

If you were to write an XML document yourself, you would in principle be free to structure it any way you wanted. But if XML documents must be exchangeable between diverse software programs, you'll probably want the XML to adhere to some rules. Because you'll not only have software producing XML but also software to do something with the XML input (like the Axiell Internet Server web application or Axiell Collections for example), and therefore the hierarchy in the XML file must be what the software expects it to be.

For Collections you usually won't write XML documents manually: they are produced internally by the software as the (intermediate and/or end) result of an export or print job or as the search result from the WebAPI. So the software will by default produce XML documents, which adhere to earlier specified rules for Axiell XML files, together forming the so-called Axiell AdlibXML* schema. Whenever third-party software produces XML documents to be processed by Axiell software at some point, it must also comply to this schema.

There are two methods to specify rules to which XML files must adhere, via a DTD (Document Type Definition) or XSD (XML Schema Definition).

- DTD is an old-fashioned way: for example, older versions of EAD (Encoded Archival Description) used it. A disadvantage of the DTD is its syntax, which allows for files to become unreadable because of their complexity.
- XSD, the XML Schema Definition is DTD's successor. It is an XML file itself.

XML used by Axiell software is formatted according to the *adlibXML.xsd* (which can be viewed in full at https://webapi.axiell.com/content/downloads/adlibXML.xsd). The most important thing you need to know about AdlibXML is that only the XML tags of the three highest levels have been defined, namely: `adlibXML` (root tag), `recordlist` (may occur only once), `record` (may occur indefinitly). Further, there is a `diagnostic` tag (in WebAPI output) on the level of the `recordlist`, which contains metadata about the search, such as the elapsed time and the number of records that were found. The fields in the records are contained within each record element and have the English field name by which they are declared in the database table *.inf* file. The structure of a Collections record itself is not defined in the schema definition because this differs per database table and XML type.

* Even though the "Adlib" brand name is no longer actively being used by Axiell, it still appears in the current AdlibXML schema name and some other legacy files and jargon still in use by Axiell software.

1.1.3 Available XML types

Within the AdlibXML schema, different XML types are possible, mainly separated into *unstructured* and *grouped* XML, but grouped XML still has variations. XML is either produced by Axiell Collections or by the Axiell WebAPI (which can process Collections data for the Axiell Internet Server web application).

■ Unstructured XML from the WebAPI or Collections

Unstructured XML can be produced by the WebAPI or be exported by Axiell Collections, if requested so.

Unstructured XML has a flat structure: all fields and their occurrences are immediate children of the `record` element. If a field has multiple occurrences, then all those occurrences are listed directly underneath each other.

One small difference between the unstructured XML from the WebAPI and Collections is that when produced by Collections, each field element also has a `tag` and `occ` attribute to provide the field tag and occurrence number, while field elements in WebAPI unstructured XML do not have these attributes. (The attributes were added to accommodate the *All fields overview (containing data) with tags* inline XSLT report in Collections, which also needed to contain field tags and occurrence numbers.)

- Of a multilingual field (if present), all language values will be exported by both the WebAPI as well as Collections. Values in the WebAPI output will have an `invariant` attribute though, while Collections output doesn't. A WebAPI example:

```
<title>
  <value lang="nl-NL" invariant="true">Mijn meertalige
titel</value>
   <value lang="en-GB" invariant="false">My multilingual
title</value>
</title>
```

  A Collections example:

```
<title tag="ti" occ="1">
  <value lang="nl-NL">Mijn meertalige titel</value>
  <value lang="en-GB">My multilingual title</value>
</title>
```

- Of an enumerative field, both the WebAPI and Collections return the neutral value plus all user interface translations listed inside the field element, in a `value` sub node per language. A Collections example:

```
<management_status tag="ms" occ="1">
    <value lang="neutral">INTERNAL</value>
    <value lang="0">internal</value>
    <value lang="1">eigen beheer</value>
    <value lang="3">Intern</value>
</management_status>
```

- The `record` element in Collections output contains a `priref`, `creation` and `modification` attribute, while in WebAPI output the `record` element contains a `priref`, `created`, `modification`, `selected` and `deleted` attribute

Below you can see an abbreviated example of a Collections export of a book record with a multilingual title field, to unstructured XML:

```
<?xml version="1.0" encoding="utf-8"?>
<adlibXML>
 <recordList>
   <record priref="3" creation="2018-10-25T10:46:40"
                      modification="2024-02-22T13:29:52">
     <priref tag="%0" occ="1">3</priref>
     <edit.date tag="dm" occ="1">2024-02-22</edit.date>
     <edit.notes tag="mm" occ="1"/>
     <edit.name tag="nm" occ="1">erik</edit.name>
     <edit.time tag="tm" occ="1">13:29:52</edit.time>
     <edit.source tag="vm" occ="1">document>book</edit.source>
     <material_type tag="ms" occ="1">Boek</material_type>
     <input.time tag="tx" occ="1">10:46:40</input.time>
     <input.date tag="di" occ="1">2018-10-25</input.date>
     <input.name tag="ni" occ="1">erik</input.name>
     <title tag="ti" occ="1">
       <value lang="en-GB">The chase</value>
     </title>
     <lead_word tag="lw" occ="1"/>
     <input.source tag="vi" occ="1">document>book</input.source>
     <material_type.lref tag="L4" occ="1">1</material_type.lref>
   </record>
 </recordList>
</adlibXML>
```

■ Grouped XML as produced by the WebAPI or Collections

Grouped XML can be produced by the WebAPI or be exported by Axiell Collections, if requested so. You can use record data in grouped XML format to create an output format, using XSLT, for example.

Grouped XML is hierarchically structured XML: fields may be a direct child of the `record` element, or when a field group name has been defined in the data dictionary, a child of a group element with the name of the group. In this case the group element is a child of the `record` element. Examples of returned XML can be studied at https://webapi.axiell.com/Topics/search.html by first adding `&xmltype=grouped` to the example queries, if needed, and then executing them.

If at least one of the fields in a field group has multiple occurrences, then the entire field group is repeated as many times. Empty occurrences of fields in a field group are retrieved as well (if specified to retrieve, in *adlibweb.xml*). The main advantage of the grouped type over the unstructured one is that it becomes easier to process repeated occurrences of grouped fields, using XSLT. In unstructured AdlibXML, all fields and field occurrences are just listed in one long list inside the `<record>` node, whilst in grouped AdlibXML, fields are grouped within a field group node (if a relevant field group exists in the data dictionary) and that field group node is repeated for each field group occurrence.

- One small difference between the grouped XML from the WebAPI and Collections is that when produced by Collections, each field element also has a `tag` and `occ` attribute to provide the field tag and occurrence number, while field elements in WebAPI grouped XML do not have these attributes. (The attributes were added to accommodate the *All fields overview (containing data) with tags* inline XSLT report in Collections, which also needed to contain field tags and occurrence numbers.
- Of a multilingual field (if present), all language values are returned as `value` subnodes of the field node; the language code and invariancy flag (the latter in WebAPI output only) per language value are returned as attributes to the value nodes.
- Of an enumerative field, both the neutral value and all available translations of the enumerative value are returned, in `value` subnodes underneath the enumerative field node; the presentation languages are attributes to the `value` nodes, and are indicated by a Collections language number (0 being English, 1 Dutch etc.), not by their language code. The presentation `language` parameter does not apply to the `grouped` XML output type.
- The `record` element in Collections output contains a `priref`, `creation` and `modification` attribute, while in WebAPI output the `record` element contains a `priref`, `created`, `modification`, `selected` and `deleted` attribute.

A partial example of grouped WebAPI output of a single record retrieved in detail:

```
<adlibXML>
  <recordList>
    <record selected="False" deleted="False"
            modification="2012-05-31T11:11:27"
            created="2007-02-07T14:40:36" priref="10">
      <acquisition.date>1816</acquisition.date>
      <administration_name>PDP</administration_name>
      <content.person.name>Venus</content.person.name>
      <content.person.name>Cupid</content.person.name>
      <content.person.name.type>
        <value lang="neutral">PERSON</value>
        <value lang="0">Person</value>
        <value lang="1">persoon</value>
        <value lang="2">personne</value>
        <value lang="3">Person</value>
        <value lang="4">شخص إسم</value>
        <value lang="6">πρόσωπο</value>
      </content.person.name.type>
      <creator.role.lref>2</creator.role.lref>
      <Dimension>
        <dimension.value>118.1</dimension.value>
```

```
        <dimension.type>height</dimension.type>
        <dimension.type.lref>6</dimension.type.lref>
        <dimension.unit>cm</dimension.unit>
        <dimension.unit.lref>8</dimension.unit.lref>
      </Dimension>
      <Dimension>
        <dimension.value>208.9</dimension.value>
        <dimension.type>width</dimension.type>
        <dimension.type.lref>7</dimension.type.lref>
        <dimension.unit>cm</dimension.unit>
        <dimension.unit.lref>8</dimension.unit.lref>
      </Dimension>
      <institution.name>The Fitzwilliam Museum</institution.name>
      <institution.name.lref>4</institution.name.lref>
      <institution.place/>
      <Material>
        <material.part>medium</material.part>
        <material>oil paint</material>
      </Material>
      <Material>
        <material.part>support</material.part>
        <material>canvas</material>
      </Material>
      <object_category>painting</object_category>
      <object_category.lref>1</object_category.lref>
      <object_number>109</object_number>
      <priref>10</priref>
      <Production>
        <creator>Palma, Jacopo il Vecchio</creator>
        <creator.date_of_birth/>
        <creator.date_of_death/>
        <creator.history/>
        <creator.qualifier/>
        <creator.role>painter</creator.role>
        <production.notes/>
        <production.place/>
      </Production>
      <Title>
        <title>
          <value lang="el-GR" invariant="false">Venus and
           Cupid</value>
        </title>
      </Title>
    </record>
  </recordList>
  <diagnostic>
    <hits>1</hits>
    <xmltype>Grouped</xmltype>
    <first_item>1</first_item>
  </diagnostic>
</adlibXML>
```

- In the grouped output, the record priref is an attribute of the `<record>` node, but appears as a separate node as well.

- Up to and including WebAPI version 3.6.1173.0, if in the grouped XML output an accessible field to be retrieved was part of a data dictionary field group, then all fields from the field group would be retrieved, even if they were empty. In later versions, only the available fields set in *adlibweb.xml* will be retrieved.
- In the grouped XML output, the names of the subnodes of a linked field are the names of the linked field in the primary database (which are the target fields for any merged-in fields).
- In the grouped XML output, the linkref field has its own subnode underneath the linked field, containing the actual linked record number.

1.2 What is XSLT

XSL(T) stands for eXtensible Stylesheet Language Transformations. It is a pattern-based language and has characteristics of programming languages as well, which you use to "transform" an XML document to some other document; this may be an XML document with the same structure but with changes made to the data in it, or it can be a differently structured XML document, or an HTML document, CSV or some other text file. During transformation, the data from the original XML document can also be processed in other ways.

Axiell Collections internally represents records as XML and when you execute an XSLT output format or display it through the *Report viewer* for the current record, this XML is passed on to the associated stylesheet which converts the XML to the desired format: this target format would usually need to be HTML if it concerns an *Inline* display format or a *Normal page* or *Raw* output (print) format, but any other desired target format (XML, plain text, etc.) is possible too if you'd like to use the resulting document for other purposes, e.g. as an exchange file to import the data into some other application. As XML-to-XML stylesheets, it allows third-party XML export files or search results to be tranformed to XML that Collections can work with, or vice versa. As XML-to-HTML stylesheets, it allows Axiell AdlibXML, like produced by the WebAPI and internally by Collections to be transformed into fully laid out pages presentable like web pages in a browser, in the Collections *Report viewer* or to be printed with a nice layout.

Originally XSLT was just named XSL, as it was thought to primarily function as layout language to produce HTML output, but as it turned out that it could be used for other transformations as well, the "T" was added. For stylesheet names it is irrelevant whether you use the extension *.xsl* or *.xslt*: there is no functional difference.

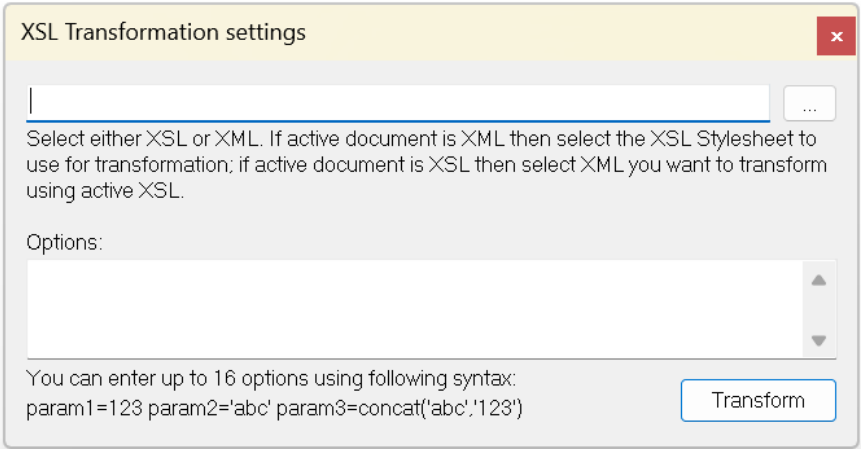
1.2.1 Applying a stylesheet to XML

You can apply a stylesheet to an XML document either:

- programmatically via the settings file of a web application like the Axiell Internet Server;
- by linking the stylesheet to your Collections application as an in-line, normal or raw output format, using Axiell Designer;
- by hardcoding a reference to the stylesheet in the XML document, like `<?xml-stylesheet type="text/xsl" href="books.xsl"?>`, but this method is insecure and is therefore not supported by most current browsers any more.
- by using a applicable code editor like Microsoft Visual Studio or Notepad++ (the latter with the *XML Tools* plugin).

In all cases a "transformation engine" does the actual transforming and produce output. Such an engine is by default part of the .NET platform and MSXML.

Using Notepad++ with the *XML Tools* plugin installed is an easy way to test your stylesheets: from an opened XML or XSLT file, select *Plugins* > *XML Tools* > *XSL transformation* to open a dialog, click the … button behind the top entry field and look up the other required file (an XSLT file for the current XML file, for instance); then click the *Transform* button to see the result.



1.2.2 A bare stylesheet

Each stylesheet starts with something like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/
 XSL/Transform">
```

and ends with: `</xsl:stylesheet>`

XSLT 3.0 and earlier versions are supported from Collections 1.14: only XSLT 1.0 was supported before that, not XSLT 2.0. Of XML, both version 1.0 and 2.0 are supported, and XSLT 1.0 can be used in an XML 2.0 document if needed.

The header may contain a third line to specify the type of output this stylesheet will generate. For HTML this is: `<xsl:output method="html"/>`

In between, you specify the actual patterns. XSLT has a syntax similar to XML, with PIs (Processing Instructions), and `<namespace:name>output</namespace:name>` elements. The namespace you always use is: `xsl`. The `names` are XSLT keywords or functions, since the `xsl` Name Space applies. XSLT is also case-sensitive.

1.2.3 XPath and templates

Suppose we have the following XML document (not AdlibXML):

```
<?xml version="1.0" ?>

<!-- my comment -->
<booklist>
  <book isbn="901234567">
    <author>Hesse, Herman</author>
    <author>Claus, Hugo</author>
    <title>Siddharta</title>
  </book>
  <book>
    <author>Wolkers, Jan</author>
    <title>Terug naar Oegstgeest</title>
    <publisher>Summer &amp; Köning</publisher>
  </book>
  <book>
    <author>Austen, Jane</author>
  </book>
</booklist>
```

XPath is similar to a path in the folder structure in Windows, but it applies to an XML document. For example, the XPath of any author in this document is */booklist/book/author*. This is relevant for the tem-

plates in your stylesheet. In XSLT, templates are the basis for the intended transformation: they contain the functions and text or HTML code to be applied, respectively added to XML elements which you consider to be a pattern. A very simple example of a stylesheet *books.xsl* (which doesn't apply any HTML codes yet) for this XML file might clarify this:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>

        <xsl:template match="/booklist">
          <xsl:apply-templates select="//author"/>
        </xsl:template>


        <xsl:template match="author">
          anonymous
        </xsl:template>

</xsl:stylesheet>
```

Two templates have been defined in here. What the transformation engine does, is it looks for template matches which it can apply (`match`), starting from the root of all XPaths. A template for the root of the XML hierarchy should always be present. Whether it can apply a template depends on whether the XPath node to match or select is accessible from the root (if the XPath to match start with a single slash) or from the current XPath level (if the XPath to match doesn't start with a slash). For example, the *author* template cannot be matched from the root, but *booklist* or */booklist* can. Each time a template gets a "match" with a node in the XML, that node becomes the current XPath level and when the matched template is done processing, the current XPath level reverts to the previous level from which the template was called with `apply-templates`. In the example above we intend to look for every occurrence of an `<author>` element in the XML file and replace it's content by the text "anonymous". So from the root node the */booklist* node is accessible, but from there the *author* node is only available outside the context of their *book* nodes if we precede it by "//": this means the *author* node can occur anywhere in an XPath.

The single forward slash (pointing to an absolute XPath starting from the root) selects only the immediate child elements of the provided node, while the double forward slash (pointing to a relative XPath) selects all descendants of the current node, regardless of their level.

The Notepad++ result of this stylesheet applied to the example XML file is the following:

```
anonymous

anonymous

anonymous

anonymous
```

If we were to leave out "//" the match could not be made, and applying the stylesheet would result in an empty page. But if you know at what level in an XPath the author node occurs you may also point directly to it, in our case via:

```
<xsl:template match="/booklist">
  <xsl:apply-templates select="book/author"/>
</xsl:template>
```

From the result you might deduce how the transformation works. There are two templates, but the `author` template cannot be matched from the root of XPath, the */booklist* can be matched though. So the transformation process enters into this template for instructions about how to transform the */booklist* node of the XML file, and this node also becomes the current XPath level. From this node we want to explicitily call the `author` template, which we do with: `apply-templates select="<relative Xpath to desired template>"`. So from the */booklist* node we can access the `author` template by selecting either *book/author* or *//author*.

And although we only call the `author` template once, it is automatically applied to all author elements in the XML file, at the selected XPath level: */author* elements placed directly underneath the */booklist* node for example, would not be matched.
In the displayed result we can also see that the titles and publisher from the XML file have been ignored; this is because we haven't specified templates for these elements yet.

By the way, if the XSLT file does exist (in the same folder), but has no templates specified, then the implicit "default" template is used to output and lay out the full XML contents.

1.2.4 Extending the stylesheet to produce proper HTML

Until now, our transformations have not produced proper HTML documents, even though we specified output method `html`. It is good prac-

tice to always adhere to the rules of the document type you are transforming to. So let's extend our stylesheet to make proper HTML.

An empty HTML file may look as follows:

```
<html>
<head>
  <title>My title for this page</title>
</head>
<body>

</body>
</html>
```

Actual content will be placed between the `<body>` tags. A simple piece of content may be:

```
<p>This is one line of <i>text</i>.</p>
```

The word "text" will be displayed in italics.

Extending our XSLT stylesheet could for example result in the following:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/booklist">
  <html>
    <head>
      <title>My title for this page</title>
    </head>
    <body>
      <xsl:apply-templates select="book"/>
    </body>
  </html>
</xsl:template>

<xsl:template match="book">
      <xsl:apply-templates select="author"/>
      <xsl:apply-templates select="title"/>
</xsl:template>

<xsl:template match="author">
  <p>
    <xsl:value-of select="."/>
  </p>
</xsl:template>
```

```
<xsl:template match="title">
  <p><i>
    <xsl:value-of select="."/>
  </i></p>
</xsl:template>

</xsl:stylesheet>
```

Note a couple of things:

- A template for the *title* node has been added.
- The XPaths to the *author* and *title* nodes are handled a little differently here. The base match now takes place on */booklist/book*.
- Instead of replacing author names by "anonymous", we display the value contained in the *author* node in the XML file, and the actual titles.
- We have added HTML tags in different places to make the output proper HTML.

The result is as follows:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html">
<title>My title for this page</title>
</head>
<body>
<p>Hesse, Herman</p>
<p>Claus, Hugo</p>
<p><i>Siddharta</i></p>
<p>Wolkers, Jan</p>
<p><i>Terug naar Oegstgeest</i></p>
<p>Austen, Jane</p>
</body>
</html>
```

This illustrates the order in which the templates have been applied. Per *book*-match, to all authors the `author` template is applied, then to all titles the `title` template. And every author and title is placed on a new line, because the HTML `<p>`-tags are in the `author` and `title` templates.

1.2.5 Using CSS stylesheets

In HTML pages you have the option to refer to a CSS (Cascading Style Sheet), although this is in no way a requirement. In a CSS you can assign font types and character layout styles to HTML structural ele-

ments (like the body of the page or tables) and to so-called layout classes which you specify yourself. The advantage of doing this in a CSS instead of just hardcoded in the HTML itself (like in the example above for the italic layout of the title), is that it is much more efficient and faster to adjust the definition of a style once, than to re-apply the adjusted style everywhere in the HTML. However, if you don't need reusable layout styles and you don't mind applying all layout through HTML tags, then you might as well leave CSS out of the equation altogether.

An example of a simple CSS is the following. Save this file as *mystyle.css* in the same folder.

```
BODY
{
  color: blue;
  background-color: lightyellow;
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 85%;
}

TABLE
{
  color: blue;
}

.title
{
  font-style: italic;
  text-decoration: underline;
}
```

Note a couple of things:

- `title` is a new class, `BODY` and `TABLE` are HTML structural elements. (The `TABLE` style will be used later on.)
- The several font types summed up behind `font-family`, indicate the priority in which these are applied. If the computer of the user doesn't have the Verdana type installed, Arial will be used, etc.
- Instead of colour names, you can also use the hexadecimal RGB (Red Green Blue) notation of colours, e.g. #DDDDDD (grey), or #ffff99 (yellow).

In an HTML document you link to a CSS in the `<head>` section:

```
<link type="text/css" href="mystyle.css" rel="stylesheet"/>
```

So in our XSLT stylesheet, where we build up an HTML page, we can do exactly the same, as can be seen in the further extended XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/booklist">
  <html>
    <head>
      <link type="text/css" href="mystyle.css" rel="stylesheet"/>
      <title>My title for this page</title>
    </head>
    <body>
      <xsl:apply-templates select="book"/>
    </body>
  </html>
</xsl:template>

<xsl:template match="book">
      <xsl:apply-templates select="author"/>
      <xsl:apply-templates select="title"/>
</xsl:template>

<xsl:template match="author">
  <p>
    <xsl:value-of select="."/>
  </p>
</xsl:template>

<xsl:template match="title">
  <p>
   <div class="title">
    <xsl:value-of select="."/>
   </div>
  </p>
</xsl:template>

</xsl:stylesheet>
```

Instead of storing the CSS code in its own file, you can also choose to include it in the XSLT stylesheet itself, in between HTML `<style type="text/css">` and `</style>` tags in the `<head>` section:

```
      <head>
        <title>My title for this page</title>
        <style type="text/css">
          BODY
          {
          color: blue;
          background-color: lightyellow;
          font-family: Verdana, Arial, Helvetica, sans-serif;
          font-size: 85%;
          }
```

```
          TABLE
          {
          color: blue;
          }

          .title
          {
          font-style: italic;
          text-decoration: underline;
          }
        </style>
      </head>
```

Without the CSS styles, you can obtain a similar result by including HTML layout tags and attributes in the XSLT templates, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/booklist">
  <html>
    <head>
      <title>My title for this page</title>
    </head>
    <body bgcolor="lightyellow">
      <font face="verdana" color="blue" size="3">
        <xsl:apply-templates select="book"/>
      </font>
    </body>
  </html>
</xsl:template>

<xsl:template match="book">
          <xsl:apply-templates select="author"/>
          <xsl:apply-templates select="title"/>
</xsl:template>

<xsl:template match="author">
  <p>
    <xsl:value-of select="."/>
  </p>
</xsl:template>

<xsl:template match="title">
  <p>
    <u><i><xsl:value-of select="."/></i></u>
  </p>
</xsl:template>

</xsl:stylesheet>
```

The HTML result of either transformation now looks as follows:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html">
<title>My title for this page</title>
</head>
<body bgcolor="lightyellow">
<font face="verdana" color="blue" size="3">
<p>Hesse, Herman</p>
<p>Claus, Hugo</p>
<p><u><i>Siddharta</i></u></p>
<p>Wolkers, Jan</p>
<p><u><i>Terug naar Oegstgeest</i></u></p>
<p>Austen, Jane</p>
</font>
</body>
</html>
```

After saving and opening this HTML file in a browser, the result looks like this:

Hesse, Herman

Claus, Hugo

Siddharta

Wolkers, Jan

Terug naar Oegstgeest

Austen, Jane

1.2.6 Applying HTML tables

Now let's try to put this in a nice table, using CSS. Again, we use standard HTML tags to accomplish this. The template and the location therein in which you place these tags matters of course. After all, do you want a table around each *author*, around each *book*, or just one for the entire *booklist*?

To get one table around all books, we have to call the `book` template within HTML `<table>` tags from within the first template and call the `author` and `title` templates within table cells and rows:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/booklist">
  <html>
    <head>
      <link type="text/css" href="mystyle.css" rel="stylesheet"/>
      <title>My title for this page</title>
    </head>
    <body>
      <table border="1">
        <xsl:apply-templates select="book"/>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="book">
  <tr>
    <td>
      <xsl:apply-templates select="author"/>
    </td>
    <td>
      <xsl:apply-templates select="title"/>
    </td>
  </tr>
</xsl:template>

<xsl:template match="author">
  <p>
    <xsl:value-of select="."/>
  </p>
</xsl:template>

<xsl:template match="title">
  <p>
    <div class="title">
      <xsl:value-of select="."/>
    </div>
  </p>
</xsl:template>

</xsl:stylesheet>
```

The resulting HTML opened in a browser looks like this:

Hesse, Herman Claus, Hugo	*Siddharta*
Wolkers, Jan	*Terug naar Oegstgeest*
Austen, Jane	

1.2.7 Functions, variables and parameters in XPath

In XSLT you can use variables but you can assign a value to it only once. So you cannot use incremental counters, or string variables which you build up piece by piece. Nor are there normal loop constructions. (The solution here is recursive programming: calling the current template from within the template, with parameters, but that is beyond the scope of this documentation.)
Let's extend the XSLT stylesheet we've been working on with some basic functionality, to finish this introduction:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="/booklist">
  <html>
    <head>
     <link type="text/css" href="mystyle.css" rel="stylesheet"/>
     <title>My books list</title>
    </head>
    <body>
      <table border="1">
        <xsl:apply-templates select="book"/>
      </table>
    </body>
  </html>
</xsl:template>
<xsl:template match="book">
  <tr>
    <td><xsl:apply-templates select="author"/></td>
    <td><xsl:apply-templates select="title"/></td>
    <td><xsl:apply-templates select="publisher"/></td>
  </tr>
</xsl:template>

<xsl:template match="author | publisher">
  <p>
    <xsl:value-of select="name()"/>
    <xsl:variable name="name">
      <xsl:choose>
        <xsl:when test="contains(., ',')">
```

```
          <xsl:value-of select="substring-after(., ',')"/>
           
          <xsl:value-of select="substring-before(., ',')"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="."/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <xsl:call-template name="printTheName">
      <xsl:with-param name="nameParameter" select="$name"/>
    </xsl:call-template>
  </p>
</xsl:template>

<xsl:template name="printTheName">
  <xsl:param name="nameParameter"/>
  :
  <xsl:value-of select="$nameParameter"/>
</xsl:template>

<xsl:template match="title">
  <p>
    <div class="title">
      <xsl:value-of select="."/>
    </div>
  </p>
</xsl:template>

</xsl:stylesheet>
```

The result looks as follows:

author : Herman Hesse author : Hugo Claus	*Siddharta*	
author : Jan Wolkers	*Terug naar Oegstgeest*	publisher : Summer & Köning
author : Jane Austen		

The first thing we may notice is that the publisher is now displayed as well. To this end we've changed the author template so that it applies to publishers too. This is done in:

```
<xsl:template match="author | publisher">
```

And in the `book` template we of course have to apply the `publisher` template as well:

```
<td><xsl:apply-templates select="publisher"/></td>
```

Then we may notice that there is "fixed" text displayed in front of authors and publishers, namely "author :" and "publisher :". "author"

and "publisher" are the names of the current XPath nodes, which you include in the output via:

```
<xsl:value-of select="name()"/>
```

In the `printTheName` template the colon is added.

But the most important change is the reversal of surname and first name. Within the `<xsl:variable name="name">` node we switch substrings. The output generated by the `select` statements is put in the `name` variable automatically simply because this output is created within the `variable` node.

In the `choose` node we have a sort of IF-THEN-ELSE, implemented here as `when` and `otherwise`. `<xsl:when test="contains(., ',')">` means if the current XML node content contains a comma, then execute:

```
<xsl:value-of select="substring-after(., ',')"/>
 
<xsl:value-of select="substring-before(., ',')"/>
```

First the current content substring behind the comma is send to output (the first name), then a space is inserted in the output (` `), then the last name is extracted and placed behind the first name and the single space. Note that functions are always put in the "value" part of a `select` statement.

If the author name or publisher name contains no comma, then no switch can be performed, so the `otherwise` part is executed: the entire node content is send to output (here, to the `name` variable).

Then the `printTheName` template is called with a parameter. The parameter `nameParameter` is filled with the value from the `name` variable; the `$` in front of `name` retrieves the value.

In the `printTheName` template first the parameter is declared. Then, in `<xsl:value-of select="$nameParameter"/>` the value from `nameParameter`, which was assigned when this template was called, is send to output (the HTML file, not the `name` variable).
Note that variables are local within a template, so the above illustrates how to pass on the value from a variable to another template.

A simpler solution might have been to output the name variable from the `author | publisher` template directly, without needing the `printTheName` template at all:

```
:<xsl:value-of select="$name"/>
```

Further note that `apply-templates` is used to apply the named template to all elements with this name in the XML file, while `call-template` calls a template which has no matching XML node.

■ Axiell Collections parameters

When Axiell Collections generates XML for output or display which will be formatted by an XSLT stylesheet, it passes a number of parameters (aka system variables) to the stylesheet. You can use these parameters and the values contained in them to enhance the functionality of your XSLT stylesheets. The available parameters are the following: namely:

- **`ui_language`** – the current user interface language as it is active in Axiell Collections. The parameter contains a standard two-letter language code, like `en` for English, `nl` for Dutch, `fr` for French, `de` for German etc. This parameter can be used in output formats.
- **`data_language`** – the currently selected data language as an IETF language tag. Examples of these IETF language codes are: `'en-GB'`, `'en-US'`, `'nl-NL'`, `'de-DE'`, `'fr-FR'`. This parameter can be used in output formats.
- **`retrievalPath`** - will contain the path or URL as set in the Axiell Designer *Retrieval path* option of an image field in the data dictionary. This parameter can be used in output formats if the path is a full URL. (So you need an image server to allow printing of images.)
- **`thumbnailRetrievalPath`** - will contain the path or URL as set in the Axiell Designer *Thumbnail retrieval path* option of an image field in the data dictionary. This parameter can be used in output formats if the path is a full URL. (So you need an image server to allow printing of images.)
  If you don't want users to be able to print your high resolution images and you have thumbnail images available in a separate folder set up in the Axiell Designer *Thumbnail retrieval path* option of an image field in the data dictionary, then you may use the `thumbnailRetrievalPath` parameter instead of the `retrievalPath` parameter.

To use the parameters in a stylesheet, declare them as a regular XSLT parameter without a default value (because it will be overwritten anyway) somewhere in the file, for example:

```
<xsl:param name="data_language”></xsl:param>
<xsl:param name="ui_language"></xsl:param>
```

It's up to you to choose which ones to use in your stylesheets. More information about these parameters and examples of their application can be found in further chapters.

1.2.8 Getting an example of the generated XML

Whenever you're about to create an XSLT stylesheet for Collections data you need to know what the generated XML looks like. In the following chapters you'll find an explanation of the different types of XML you can expect and many examples, but if you're still unsure and there's no obvious way to view the generated XML, you may create the following very small stylesheet to output the actually generated XML without transforming it to anything else, giving you a good example to work with:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" ver-
sion="1.0">
  <xsl:template match="/">
    <textarea rows="70" cols="90">
      <xsl:copy-of select="/" />
    </textarea>
  </xsl:template>
</xsl:stylesheet>
```

or, using a different method, to output the generated XML within the base `<adlibXML>` node, as part of an empty HTML document:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" vers-
ion="1.0">
  <xsl:output method="html" />
  <xsl:template match="/adlibXML">
    <html>
      <head>
        <title>Get Axiell Collections output XML</title>
      </head>
      <body>
        <xmp>
          <xsl:copy-of select ="*"/>
        </xmp>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

1.2.9 Best practice

You should always start by creating a template (match) for the XPath root node, in the example above this is */booklist*. This is your main template from which you should select other templates when appropriate.

You could arrange all other templates by size, the smallest ones at the bottom of the stylesheet, increasing in size towards the top. But a grouping based on "procedural" templates versus non-procedural ones would also make sense.
Template sizes should be kept as small as possible. Preferably, any template should be able to fit on your screen entirely. You can achieve this by optimizing any functionality.

For transformation to HTML, it is recommended to use a CSS to specify character layout styles like fonts and colours. This keeps you XSLT stylesheet more clean, and changes in layout styles are easier to implement in a CSS stylesheet anyway.

Comment your XSLT stylesheet as much as possible, with: `<!-- my comments -->` Comments cannot be nested, so if you want to "comment out" a large piece of code which already has comments in it, use a `when test="0"` around it.

1.2.10 Other uses of XML and XSLT

Through a so-called gateway it's possible to restructure queries made in Collections to fit the syntax of third-party database software. The gateway then accesses such a database over the internet, for instance via HTTP or through SRU. When the search result comes back as XML, it is probably not AdlibXML. However, by using XSLT stylesheets in the gateway, it is possible to transform the foreign XML to AdlibXML, which is then send back to the Collections application where the data is ready to be derived into the Collections database. This way, foreign databases can be accessed as if they were "friendly" Collections databases.

1.2.11 More information

For more information about XPath, see a third-party manual or the internet, for example: http://www.w3.org/TR/xpath

2 Creating output formats

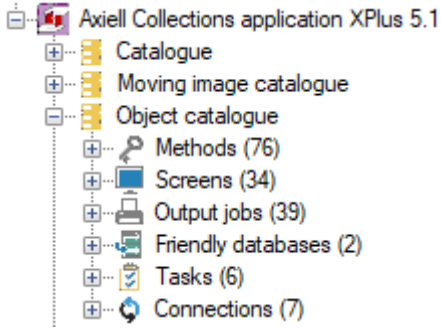
2.1 Grouped XML for XSLT export/output formats

When you have marked one or more records in Axiell Collections, you can choose to print them via a standard or custom output format. In Axiell Collections you can use the printer icon in the top toolbar, to print either all records from the result set, all marked records or just the currently selected record.

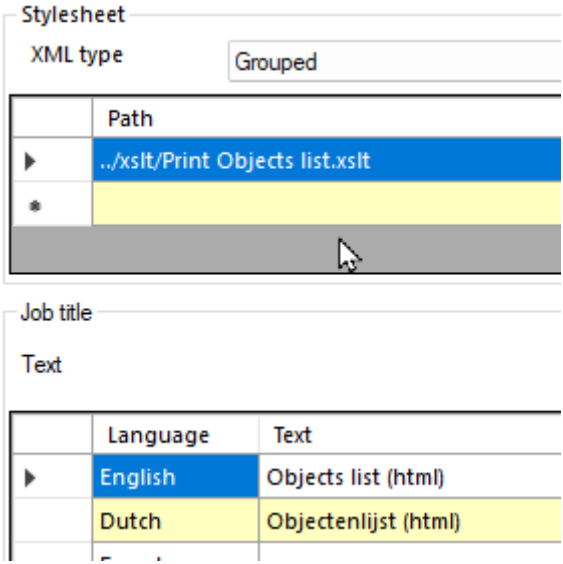
One way to create a custom output format is to build an appropriate XSLT stylesheet. Axiell Collections internally processes records as XML and when you apply an XSLT output format, this XML is passed on to the stylesheet which converts the XML to the desired format: this target format would need to be HTML if it concerns a print format, or any other desired format (XML, csv, etc.) if you mean to use the output as an export file.
Axiell Collections can generate either unstructured XML or grouped XML. Which XML type must be generated by Collections, can be set per XSLT *Output job* via Axiell Designer.

2.1.1 Setting the XML type in Designer

Output jobs (aka output formats or print formats), are registered per data source (like the *Object catalogue* for example) underneath an application definition (like that of *Xplus 5.1* for example) in the Application browser of Axiell Designer.



The XML type for an output job can be set in the *XML type* option on the *Output job properties* tab of a selected output job. (See the Designer Help for more information about setting up output jobs.)



2.1.2 Advantages of grouped XML for use in stylesheets

The main advantage of the grouped type over the unstructured one is that it becomes easier to process repeated occurrences of grouped fields. In unstructured AdlibXML, all fields and field occurrences are just listed in one long list inside the `<record>` node, whilst in grouped AdlibXML, fields are grouped within a field group node (if a relevant field group exists in the data dictionary) and that field group node is repeated for each field group occurrence.
Whenever you create an XSLT stylesheet for unstructured AdlibXML, which must be able to collect field data per field group occurrence, you have no choice but to always count the "position" of every processed field occurrence because that's the only way to retrieve the other fields from the same position. In grouped AdlibXML on the other hand, there's no need for such a workaround because every field group occurrence is contained within its own field group node. Matching an XSLT template to a field group node automatically provides access to all grouped fields with the same occurrence number (in other words: at the same position).

■ Examples of working with occurrences

Suppose you wish to create an output format based on an XSLT stylesheet, to print the object name(s) and the notes pertaining to the object name, of a museum object. The `object_name` and `object_name.notes` fields, as specified in the data dictionary of the *Collect* database table, are part of a field group called `Object_name`. Be-

cause of this grouping you can repeat these two fields (and the others belonging to the group) as a group in the Collections record. When you print these group repetitions, you will want to keep them grouped of course: you don't want a list of all object names followed by a list of all notes.

For unstructured AdlibXML you would have to tackle this problem as follows:

```
<?xml version="1.0" encoding="utf-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="html" />
    <xsl:template match="/adlibXML">
      <html>
        <head>
          <title>Field group handling for unstructured XML</title>
        </head>
        <body>
          <xsl:apply-templates select="recordList/record"/>
        </body>
      </html>
    </xsl:template>

    <xsl:template match="record">
      <xsl:apply-templates select="object_name"/>
    </xsl:template>

    <xsl:template match="object_name">
      <xsl:variable name="pos" select="position()" />
      <p>
        <xsl:value-of select="."/>
        <br/>
        <xsl:apply-templates select="../object_name.notes[$pos]"/>
        <br/>
      </p>
    </xsl:template>

    <xsl:template match="object_name.notes">
         <xsl:value-of select="."/>
    </xsl:template>

  </xsl:stylesheet>
```

For grouped AdlibXML on the other hand, you could code this as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
    <xsl:output method="html" />
    <xsl:template match="/adlibXML">
      <html>
        <head>
```

```
          <title>Field group handling for grouped XML</title>
        </head>
        <body>
          <xsl:apply-templates select="recordList/record"/>
        </body>
      </html>
    </xsl:template>

    <xsl:template match="record">
      <xsl:apply-templates select="Object_name"/>
    </xsl:template>

    <xsl:template match="Object_name">
      <p>
        <xsl:apply-templates select="object_name"/>
        <br/>
        <xsl:apply-templates select="object_name.notes"/>
        <br/>
      </p>
    </xsl:template>

    <xsl:template match="object_name">
      <xsl:value-of select="."/>
    </xsl:template>

    <xsl:template match="object_name.notes">
         <xsl:value-of select="."/>
    </xsl:template>

  </xsl:stylesheet>
```

The output of either stylesheet is structured like this:

object name in field group occurrence 1 of record 1
object name notes in field group occurrence 1 of record 1

object name in field group occurrence 2 of record 1
object name notes in field group occurrence 2 of record 1

object name in field group occurrence 3 of record 1
object name notes in field group occurrence 3 of record 1

…

object name in field group occurrence 1 of record 2
object name notes in field group occurrence 1 of record 2

object name in field group occurrence 2 of record 2
object name notes in field group occurrence 2 of record 2

…

2.2 Printing images

The image reference (aka reproduction reference) in your Collections records does usually not consist of a full path to an image file. In current application versions, by default a storage/retrieval path has been set for the image field, so that only the image file name is present in the image reference field. Since the HTML output we would like to generate requires a URL to retrieve an image, we need to find a way to combine the image file name from the image reference field with the URL to the images folder (a file system path won't do). A WebAPI call to an image server, as is often used as storage/retrieval path for image fields in Axiell Collections is exactly what we need. To get this base URL in your stylesheet, you can hard code it in there and concatenate it with the image file name, like in the following example (for grouped XML):

```
<xsl:template match="Media">
  <xsl:variable name="imageFileName">
      <xsl:value-of select="media.reference"/>
  </xsl:variable>
  <xsl:variable name="imagePath">
      <xsl:text>http://ourserver.com/images/wwwopac.ashx?
command=getcontent&amp;server=images&amp;value=</xsl:text>
      <xsl:value-of select="$imageFileName"/>
  </xsl:variable>
  <p>
      <img border="0" width="340" src="{$imagePath}" />
  </p>
</xsl:template>
```

As you can see, this template matches the `Media` field group. It fills a new `imageFileName` variable with the (first) linked image file name from the `media.reference` field. Next, another new variable named `imagePath` is created and filled with the base URL to our image server after which the image file name is pasted behind it. And finally the contents of the `imagePath` variable is used as the `src` attribute of the HTML `img` tag (to retrieve the image in the resulting HTML page).

However, you can also use the `retrievalPath` or `thumbnailRetrievalPath` parameter to dynamically retrieve the URL, instead of hard coding it but then you'll have to strip off the `%data%` part of that URL. When you print selected records from Collections using an XSLT output format, Collections will pass the relevant path in the appropriate parameters to the stylesheet. At the top of your stylesheet you would declare the parameter:

```
    <xsl:param name="retrievalPath"/>
```

And you would need templates like the following:

```
<xsl:template match="media.reference">
    <xsl:variable name="imagePath">
      <xsl:call-template name="replace-string">
        <xsl:with-param name="text" select="$retrievalPath"/>
        <xsl:with-param name="replace" select="'%data%'"/>
        <xsl:with-param name="with" select="."/>
      </xsl:call-template>
    </xsl:variable>
  <p>
  <img border="0" height="180" src="{$imagePath}" />
  </p>
  <p>
  </p>
</xsl:template>

<xsl:template name="replace-string">
  <xsl:param name="text"/>
  <xsl:param name="replace"/>
  <xsl:param name="with"/>
  <xsl:choose>
    <xsl:when test="contains($text,$replace)">
      <xsl:value-of select="substring-before($text,$replace)"/>
      <xsl:value-of select="$with"/>
      <xsl:call-template name="replace-string">
        <xsl:with-param name="text"
         select="substring-after($text,$replace)"/>
        <xsl:with-param name="replace" select="$replace"/>
        <xsl:with-param name="with" select="$with"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$text"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

For an enterprise solution, in which images for the different branches are stored in their own folders, you can still use a single image server: with the `<folderMappingsList>` settings in adlibweb.xml you can specify these different folders. In such case you need to extend your WebAPI call with the `folderId` parameter which must be assigned the record number of the currently processed, linked media record. The first example above can then be adapted to the following:

```
<xsl:template match="Media">
  <xsl:variable name="imageFileName">
      <xsl:value-of select="media.reference"/>
  </xsl:variable>
  <xsl:variable name="imageRecordLref">
      <xsl:value-of select="media.reference.lref"/>
  </xsl:variable>
```

```
  <xsl:variable name="imagePath">
      <xsl:text>http://ourserver.com/images/wwwopac.ashx?
command=getcontent&amp;server=images&amp;value=</xsl:text>
      <xsl:value-of select="$imageFileName"/>
      <xsl:text>&amp;folderId=</xsl:text>
      <xsl:value-of select="$imageRecordLref"/>
    </xsl:variable>
    <p>
      <img border="0" width="340" src="{$imagePath}" />
    </p>
</xsl:template>
```

2.3 Retrieving multilingual data

To retrieve data from a multilingual field, a *title* field for example, in the current data language, you would use code fragments like:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="html" />
  <xsl:param name="data_language"/>

  <xsl:template match="/adlibXML">
…
        <xsl:apply-templates select="recordList/record"/>
…
  </xsl:template>

  <xsl:template match="record">
…
    <xsl:apply-templates select="Title/title/value
    [@lang=$data_language]|Title/title/value
    [@lang='']"/>
…
  </xsl:template>


  <xsl:template match="Title/title/value">
    <xsl:value-of select="."/>
    <br />
  </xsl:template>
…
```

Or, to retrieve and display maximally four specific data language values of all occurrences of the *title* field of the current record and precede each value by the proper language code, you can use the following parameter declaration, template call and actual template:

```
<xsl:param name="data_language"/>

<xsl:apply-templates select="Title/title/value"/>
```

```
<xsl:template match="Title/title/value">
    <p>
        <xsl:choose>
          <xsl:when test="@lang='nl-NL'">
            <xsl:text>nl-NL: </xsl:text>
          </xsl:when>
          <xsl:when test="@lang='en-GB'">
            <xsl:text>en-GB: </xsl:text>
          </xsl:when>
          <xsl:when test="@lang='fr-FR'">
            <xsl:text>fr-FR: </xsl:text>
          </xsl:when>
          <xsl:when test="@lang='de-DE'">
            <xsl:text>de-DE: </xsl:text>
          </xsl:when>
          <xsl:otherwise>
            <xsl:text>&lt;missing value&gt;</xsl:text>
          </xsl:otherwise>
        </xsl:choose>
          <xsl:value-of select="."/>
    </p>
  </xsl:if>
</xsl:template>
```

2.4 Using a page break

At the end of a record node you could use the following code to force a page break after every two records. (You can change to number 2 to a different number if you want a page break after a different number of records.)

```
    <xsl:if test="position() mod 2 = 0">
      <p style="page-break-before:always" />
    </xsl:if>
```

2.5 Printing barcode labels to a normal printer

This chapter offers an example of an XSLT stylesheet (made for grouped XML) to print simple barcode labels to a normal printer from within Axiell Collections. You can print to paper or label sheets. The example is really just that, a very basic example to show you how you can build such stylesheets yourself.

```
<?xml version="1.0" encoding="utf-8"?>
 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
   <xsl:output method="html" />
   <xsl:template match="/adlibXML">
     <html>
       <head>
```

```
        <title>Object number barcode</title>
        <style type="text/css">
           .text
          {
          font-family: Verdana;
          font-size: large;
          vertical-align: top;
          }
          .titletext
          {
          font-family: Verdana;
          font-weight: bold;
          font-size: large;
          }
          .innertable
          {
          border: solid 0px;
          border-collapse: collapse;
          }
          @font-face {
          font-family: "Free 3 of 9 Regular";
          src: url(free3of9.ttf) format('truetype');
          }
          span.barcode
          {
          font-size:48pt;
          font-family: "Free 3 of 9 Regular"
          }
        </style>
      </head>
      <body>
        <xsl:apply-templates select="recordList/record"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="record">
   <table width="500px" border="0" cellspacing="0" cellpadding="5"
    class="innertable">
    <tr valign="top">
     <td align="center" class="text">
      <xsl:apply-templates select="object_number" mode
="barcode"/>
     </td>
    </tr>
    <tr valign="top">
     <td align="center" class="text">
      <xsl:apply-templates select="object_number" mode ="text"/>
      <p>_____</p>
     </td>
    </tr>
   </table>
  </xsl:template>

  <xsl:template match ="object_number" mode ="barcode">
```

```
    <span class="barcode">
      <!-- Barcode 39 needs a start and stop character * -->
      <xsl:text>*</xsl:text>
      <xsl:value-of select "."/>
      <xsl:text>*</xsl:text>
    </span>
  </xsl:template>

  <xsl:template match ="object_number" mode ="text">
      <xsl:value-of select "."/>
  </xsl:template>

</xsl:stylesheet>
```

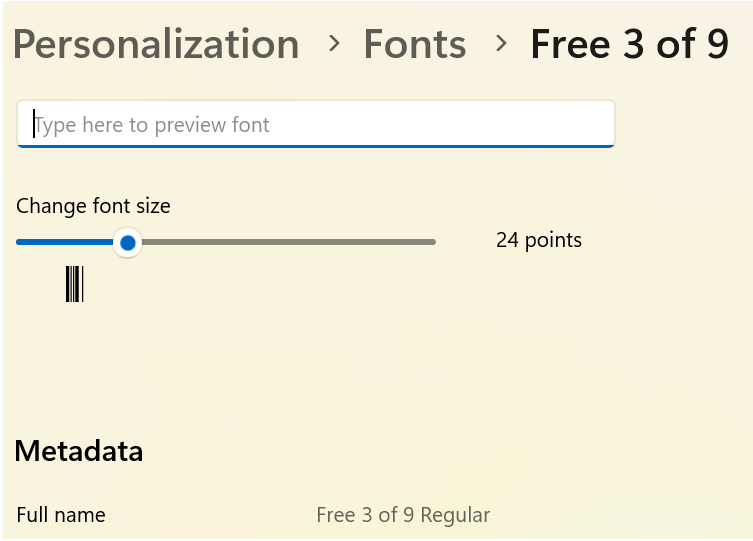
You can change it all you like of course, and to get it working in Axiell Collections you do indeed have to make some changes to it.

You have to have a barcode font in the *.ttf* or *.woff* format available on a web server on the same domain as Collections or present in the same folder as the generated HTML file, AND the font needs to have been installed in Windows. If you haven't got a barcode font, you'll have to purchase such a font first, find a freely available font on the internet or try to install a demo version of a suitable font. You refer to this font in the CSS section of the stylesheet:

```
@font-face {
font-family: "Free 3 of 9 Regular";
src: url(free3of9.ttf) format('truetype');
}
span.barcode
{
font-size:48pt;
font-family: "Free 3 of 9 Regular"
}
```

Replace the URL by your own URL. In this example, the font file should be present in the same folder as the generated HTML file

Important to note is that the font-family name has to be exactly the same as the *Full name* of the font. You can find this name in the Windows font settings.



A limitation of this implementation is that the HTML won't be portable as far as the barcode is concerned. When printing with this stylesheet, HTML will be generated. Often you will print this HTML directly and the barcode will be printed too, but if you ever copy the HTML itself to save it in a file or send it by e-mail you'll notice that it can't show the barcode when that HTML is opened again, outside of the Collections session. (It will then show the number instead of the barcode.)

2.6 Creating text labels from HTML fields

An HTML field is a database field meant for long, laid-out text, possibly including images, much like a small and simple web page. Layout can be applied to the text during editing of the record. You could use such a field to create printable text labels to be presented with your museum objects, for example. From within Axiell Collections you can print the contents of such a field to a Word template or with the aid of a custom XSLT stylesheet, whilst keeping the layout intact. Although normally you will only see the laid-out text (not the HTML code) while you are editing an HTML field, the field contents will actually be stored as (editable) HTML code in the background.

For example, in current application versions you can find a *label.text* field (tag AB) on the *Accompanying texts* tab and an *inscription.content* field (tag TR) on the *Inscriptions* | *Markings* screen in the Object catalogue, both HTML fields.

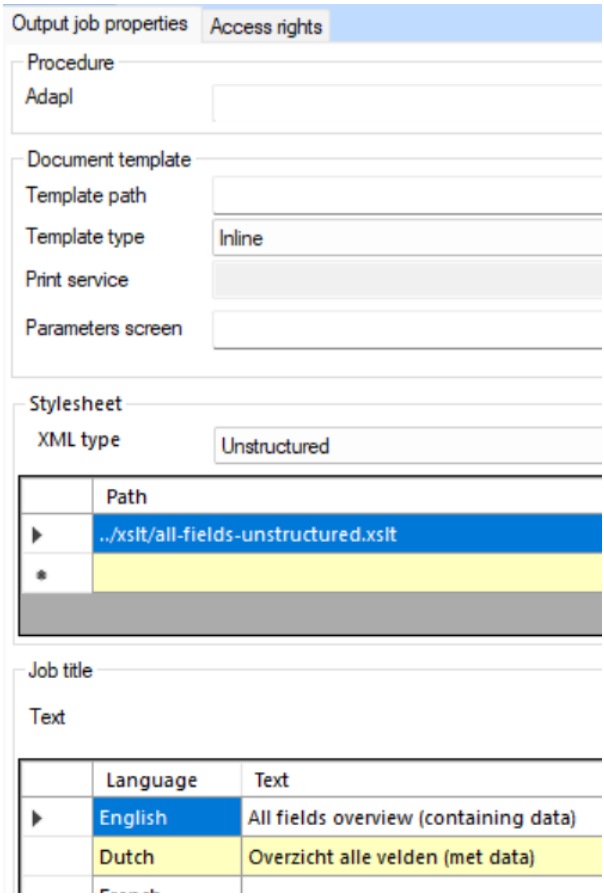
The HTML code in these fields has no `<body>` tags or `<head>` section, it may just begin with a `<p>` tag and end with a `</p>` tag. If you add `<body>` tags or a `<head>` section anyway, then these will be removed as soon as you click *OK*. When you click *OK*, the HTML will also be adjusted automatically to allow it to be saved within the (XML) storage format of the Collections record. Any indentation and layout of the HTML code itself will be removed as well, so that the code will be displayed as a single paragraph of text next time you open the editor; since such a cluttered presentation makes it hard to edit the code, we recommend to keep the code relatively short.

When records with HTML fields are output as XML using Collections or the WebAPI, the HTML field contents will be extracted as HTML code within the XML field tags, where XML reserved characters like < and > in the HTML code will have been converted to `&lt;` and `&gt;`.
For both printing and browser display, the escaping of these characters needs to be reversed so that the resulting HTML contains the < and > characters again. This is done with the `disable-output-escaping="yes"` attribute for the `xsl:value-of` element in the XSLT stylesheet. Then a template to retrieve the HTML contents correctly for printing and display doesn't need to be more than the following:

```
<xsl:template match="label.text">
  <xsl:value-of select="." disable-output-escaping="yes"/>
</xsl:template>
```

3 Inline reports for the Report viewer

The *Report viewer* in Collections offers the user a different view of the contents of the currently selected record, a raw list of all fields with data maybe or a nicely laid out compact presentation of some of the record contents. XSLT output formats set up in Designer with the *Template type* property *Inline*, will be available both as a regular output formats (for multiple marked records) as well as an inline report in the *Report viewer* for the currently selected record only.



Current Collections applications already offer a few of these inline reports, but you can add your own as well. To program them is not different from programming a normal XSLT output format.

Inline reports for the Report viewer

An inline report would be perfectly suited to employ the `ui_language` parameter, to show fixed texts in the current Collections interface language. And while we're at it, for presentation purposes it would also be nice if the formal *surname, first name* format of the registered creator name would be reversed to the informal *first name surname* format. In the following example you can see how to tackle this if you'd like to display a Dutch fixed text if the interface language is Dutch (nl) and an English text for all other interface languages:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" encoding="UTF-8" indent="yes"/>
  <xsl:param name="ui_language"></xsl:param>

  <xsl:template match="/adlibXML">
    <html>
      <head />
      <body>
        <font face="verdana" size="3">
          <xsl:apply-templates select="recordList"/>
        </font>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="recordList">
    <xsl:apply-templates select="record"/>
  </xsl:template>

  <xsl:template match="record">
    <xsl:apply-templates select="Production"/>
  </xsl:template>

  <xsl:template match="Production">
    <xsl:variable name="pos" select="position()"/>
    <xsl:choose>
      <xsl:when test="$pos = 1">
        <xsl:choose>
          <xsl:when test="$ui_language = 'nl'">
            <xsl:text>Vervaardigd door </xsl:text>
          </xsl:when>
          <xsl:otherwise>
            <xsl:text>Created by </xsl:text>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text> &amp;amp; </xsl:text>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:apply-templates select="creator"/>
  </xsl:template>
```

```
  <xsl:template match="creator">
    <xsl:choose>
      <xsl:when test="contains(., ',')">
        <xsl:value-of select="substring-after(., ', ')"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="substring-before(., ',')"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="."/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```