A Beginners Guide XRX

A step-by-step guide to creating your first XRX application with eXist. Version 4.

Dan McCreary <`dan@syntactica.com`>
Joe Wicentowski <`joewiz@gmail.com`>

A Beginners Guide XRX: A step-by-step guide to creating your first XRX application with eXist. Version 4.

by Dan McCreary and Joe Wicentowski

Table of Contents

List of Figures

List of Examples

Chapter 1. Getting Started

Introduction

The following is a is a Beginner's Guide for creating a new XRX application in the eXist application server. Beginners Guides are intended for people that are new to eXist and are interested in building their first web applications. This Beginners Guide demonstrates the minimal code necessary to perform CRUDS operations. These are operations are **C**reate, **R**ead (or view), **U**pdate, **D**elete and **S**earch. Creating your first XRX application can be somewhat tricky since there are several structures that need to be "wired" together correctly for the CRUDS components to work correctly. This example tries to use as little code as possible and yet still cover many of the key components of a fully functional XRX web application.

Intended Audience

Creating a new web application from scratch is a core skill that is necessary to understand the power of the XRX web application architecture. Our experience has shown that once users get an understanding of how XRX applications are constructed they can quickly become productive building new web applications. They also have a much better understand of the complex portions of the XRX application and why these portions are usually automated in XRX frameworks.

This document is designed for new eXist users that would like to create their first XRX application. For this process we assume that you have a basic understanding of XML and understand concepts such as XML elements and XPath expressions. The user should also be somewhat familiar with very basic HTML markup including the structure of an XHTML file and use of HTML lists and HTML tables. We will also be describing how XQuery is used to create listing of items and viewing an individual item. Users should review the basic structure of a FLOWR expression (for, let, order, where return) and basic XQuery syntax. Familiarity with the fundamentals of XForms is helpful but this guide will explain each of the XForms elements used in the example.

We should note that there are several easy-to-use drag-and-drop GUI tools available that can create XForms and there are systems that can also automatically create a fully functional XRX application directly from an XML Schema. But using these tools and frameworks hide much of the inner workings of building an XRX application. So this tutorial is for those that want to have a clear understanding of how XRX systems work.

Getting Started

To use this guide you will need to have following tools in place:

1. **eXist:** You will need to have a version of eXist running on your local system. By default eXist runs on port 8080 so that when you set your web browser to http://localhost:8080 you should see the eXist homepage come up. You can also run this tutorial on any remote server. If you are doing this you must replace the word localhost with the name of your remote server. Make sure that you take into account the port number. The might not be 8080 and it may be missing which means that port 80 is implied.

2. **Editor:** You will need some tool to edit XML and XQuery files. We strongly encourage you to use a tool such as the oXygen XML Editor since this tool has special additions to make editing XML and XQuery files easy. Simple text editors such as Microsoft Notepad will work but will not give you immediate feedback when there are syntax errors in your files. Syntax highlighting is very useful when you are first learning an new programming system. Since there are 30-day free trials of many tools we strongly encourage new users to use good XML editors.

3. **Uploader:** You will need some tool to transfer your files directly to eXist. Tools like oXygen can save directly to the eXist database or you can also use a WebDAV client to copy the files. There are also web uploader tools in the eXist admin area and there are custom versions that also allow

you to upload and expand an entire ZIP file within the database. As a final option you can also use the eXist Java console to upload files.

4. **XForms:** You will need some XForms client libraries. This example will use the XSLTForms client which is usually installed in /db/xforms/xsltforms. You do not have to use libraries that are in the eXist database but this is sometimes preferable.

Terms and Concepts Used

This example will use the following terms and concepts:

XRX: XRX is the name of the web application architecture that we will be using in this example. XForms are used in the client (web browser), REST is the style of interfaces to the database and XQuery is the server language. The most significant portion of XRX is that it does not require the users to translate data into Java or .Net objects and it will never require that the user "shred" documents into rows of a relational database.

XForms: a set of around 25 XML tags that are used to define the structure of a web from. XForms is much more advanced then traditional HTML forms but requires first time user to bind user interface controls to each leaf element in an XML instance. XForms stores the data in a model element in the HTML HEAD tag and then binds the leaf elements in the model to web input controls. Most simple forms need only a few control types and these can be quickly learned.

XQuery: the query language for selecting XML structures from the XML database. XQuery is a little different then other languages you may have used in the past. It is a "functional" programming language that makes it very easy to create robust server-side programs that do not have many of the "side-effects" of other languages. It is similar to the SQL language in some ways but it is specifically designed for selecting and transforming XML documents. Because of the indexing structure of eXist it is also very fast, even when working with gigabytes of XML data.

Note for New XQuery Users

There are some things that are very different in XQuery that you should be aware of. In general, all XQuery variables are *immutable*, meaning that they are designed to be set once but never changed. So functions like let $x := $x + 1 within loops will not increment like in procedural languages. There are also restrictions on what can be done inside FLOWR statements. We will illustrate these in examples later in other Beginner's Guides.

REST: the term we use to describe that many of the parameters to our XRX application will be done by simply placing parameters at the end of a URL. For example to pass a query keyword to a our XRX application search service the URL appends search.xq?q=myword. This means you just need a web browser to test services. No complex SOAP interface testing tools are required. To execute any XQuery program that is running in the eXist database you simply enter the URL to that function in the web browser. For example the home page of the test application under the default configuration will be http://localhost:8080/exist/**rest**/db/apps/term/index.html. Note that the word "rest" comes after the /exist/ and before the /db/.

WebDAV: the term we use to describe how bulk files are moved to and from eXist and how files are listed within eXist collections. If you want to add a folder to eXist you can do this through the WebDAV interface. When you use oXygen or other editors you will also use the WebDAV interface. To open a file through the WebDAV interface you might open http://localhost:8080/exist/**WebDAV**/db/apps/term/.

Model-View-Bindings: the term we use to describe how user interface elements (controls) within a form are associated with leaf-level elements within the XForms model. This is similar to the Model-View-Controller (MVC) architecture in other systems but in the case of XForms is that event controls are part of the views. By using XPath statements in the **ref** attributes for user interface controls a dependency graph is constructed that keeps the model and views in sync. This makes forms development much easier since the form developer never needs to manually move data between the model and the views.

Convention over Configuration: the process of using standardized collection and file names for frameworks to be automated across all XRX applications. Users have the ability to change these conventions but they are then responsible for maintaining their own frameworks. The reason for using a generic file name such as `list-items.xq` instead of `list-terms.xq` may not be clear to you at first, but as you will see later, this more general file naming convention has it merits when many applications are managed.

Collection and File Conventions

When we build an XRX application it is important to create a set of collections that will help us structure and our application. Although you do not have to use the collection conventions used in this example, you will find that many frameworks that use this "convention" will be much easier to build and maintain. The philosophy of convention over configuration is implicit in this design. You are always free to change the names of the collection or the queries but you must take full responsibility of building your own frameworks if you vary from these conventions.

Here are the standards we strongly recommend you use for your first application:

1. **Apps:** All XRX applications should be grouped in a single collection. For example `/db/apps` or `/db/org/mycompany/apps`. The exact location of the apps collection in the databases in not relevant but all apps should be stored together in a collection called apps.

2. **App:** Each XRX application should be grouped in a collection. This collection name should reflect the function of the application. For example our business term manager might be `/db/apps/terms`. The convention is to use the plural (terms not term) if the application manages multiple business terms.

3. **Data:** Each XRX application should store its data in a separate data collection. For example our term manager application will store all the data in /db/apps/terms/data. In this example the first term will be stored in the file `1.xml` and the second in the file `2.xml` etc. When the user saves new terms we can increment a counter to add a new term.

4. **Views:** Each XRX application should store read-only views of the data in a views collection. In our example our term manager will store read-only views of the data in the `/db/term/apps/terms/views`. Note that views are read-only and functions that do not alter the XML data. Tools that change or edit the data are not usually stored in the views collection. This allows access control system to limit who change or delete data.

5. **Edit:** Each XRX application should store its edit function in a collection called edit. For our term manager application this would be /db/apps/term/edit. Edit function include saving new terms, updating terms and deleting terms. By grouping all edit functions together it is easy to deny access to users that do not have permission to change items.

6. **Search:** Each XRX application should store its search functions in a collection called `search`. For our term manager application this would be /db/apps/term/search. There are two functions stored here. A simple HTML search form (search.html) and a RESTFul search service.xq. Advanced applications sometimes combined these functions into a single XQuery that generates HTML. In addition to these two search function an additional configuration file must be stored in the /db/system/config/db/apps/terms/data collection that describes how the files are indexed for search.

7. **AppInfo:** Each XRX application should store information that pertains to the application in an XML file within the main application collection. By convention this file is called the app-info.xml file. Information such as the application name, description, author, version, license, dependencies etc. should be stored in this file. This tutorial will not cover this file structure but you may see it in many of the sample programs. This will be covered in other XRX Beginners Guides.

Example Data: Business Terms

In this example we will use a simple registry of business terms that might be used in a glossary of terms on a web site. Each term will have a term name, a definition and a publish-status code of draft, under-review or published.

Example 1.1, “Sample Structure” is a sample structure of the XML file for a sample term:

Example 1.1. Sample Structure

```
<term>
    <id>1</id>
    <term-name>Declarative Programming</term-name>
    <definition>A style of programming that allows users to declare their requi
        want done) and leave out the details of how the function should be perf
    <publish-status-code>published</publish-status-code>
</term>
```

In this example we have selected data items that will use a simple input field for the name, a textarea for the definition and a selection list for the status codes.

Views

There are two XQuery services we will create in our initial XRX application. One is a simple XQuery that will list all the terms in our data collection that have the root element `term`. This file is called `list-items.xq`. The second is an XQuery function that displays all of the elements individual term in HTML format. We call this `view-item.xq`. The `view-item.xq` XQuery requires a single parameter which is the ID of the term. These queries will allow the user to drill down to see an individual terms by first viewing a list of all the term in a collection.

Listing Items

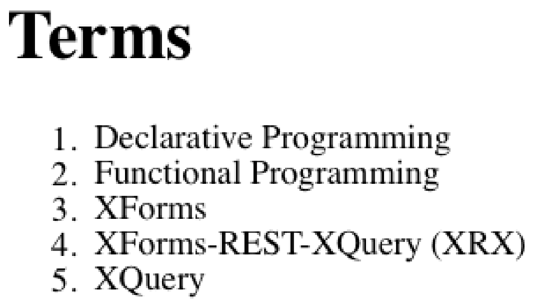
Our first task will be to create a simple XQuery program that will list all the terms in our collection in an HTML file. To do this we will us a simple XQuery FLOWR loop that gets each of there terms in the collection in succession and then converts the XML into a HTML list item using the `<li>` tag. The convention to use in Example 1.2, “`/db/apps/terms/views/list-items.xq`” is the file name `list-items.xq`.

Example 1.2. `/db/apps/terms/views/list-items.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";
<html>
   <head>
      <title>Glossary of Terms</title>
    </head>
    <body>
       <h1>Terms</h1>
       <ol>{
         for $term in collection('/db/apps/terms/data')/term
            let $term-name := $term/term-name/text()
            order by $term-name
            return
               <li>{$term-name}</li>
      }</ol>
    </body>
</html>
```

Figure 1.1, “Output from `list-items.xq`” shows the output:

Figure 1.1. Output from `list-items.xq`



Terms

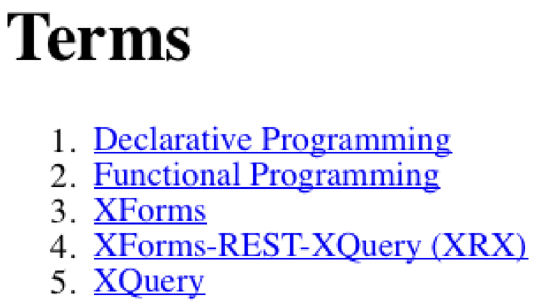
1. Declarative Programming
2. Functional Programming
3. XForms
4. XForms-REST-XQuery (XRX)
5. XQuery

There are few items to note. First we that we use the collection function to specify what data is being listed. We also return only items in the data collection that have term as their root element. This allows us to put other data types within the data collection without disrupting this report.

Our next step is to change each of the items listed into HTML links so that we can view each individual item on a separate HTML page. To do this we change the `<li>{$term-name}</li>` to be the following code:

```
<li><a href="view-item.xq?id={$term/id/text()}">{$term-name}</a></li>
```

This produces the following output:



Terms

1. Declarative Programming
2. Functional Programming
3. XForms
4. XForms-REST-XQuery (XRX)
5. XQuery

This has the effect of taking the ID out of each term and passing it as a RESTful parameter to out next query that will view each term. We will use this same technique many times. Note that this uses a relative path to the view-item.xq program. So it is important to keep both the list items and the view items in the same collection for this to work correctly.

Note that this list items works fine as long as we have just a few hundred terms. But as your collections get longer (usually above a few hundred items) you will want to create a list items query that only lists the first 30 or so items and then has a NEXT button to get more items. This will be covered in another section. (see Pagination in XQuery)

Viewing an Individual Item

Now that we have a list of all the items in a collection we are ready to drill down to a specific item and see all of the information about a single item. By convention this is done by an XQuery file call "view-item.xq". The item viewer takes a single parameter that is the ID of the item. It has to perform a query on all the items in the data collection to find only the item you are looking for. This is done by adding a "predicate" or "where clause" to the query. In general the predicate will be used because it will be faster. The structure of the line that gets an individual item (for example term id=5 from a collection is the following:

```
let $term := collection('/db/apps/terms/data')/term[id='5']
```

Note that the predicate [id='5'] indicates to the system that only a term with an ID of 5 should be returned. Also note that we are doing simple string comparison and we are not converting the ids into integers in this example.

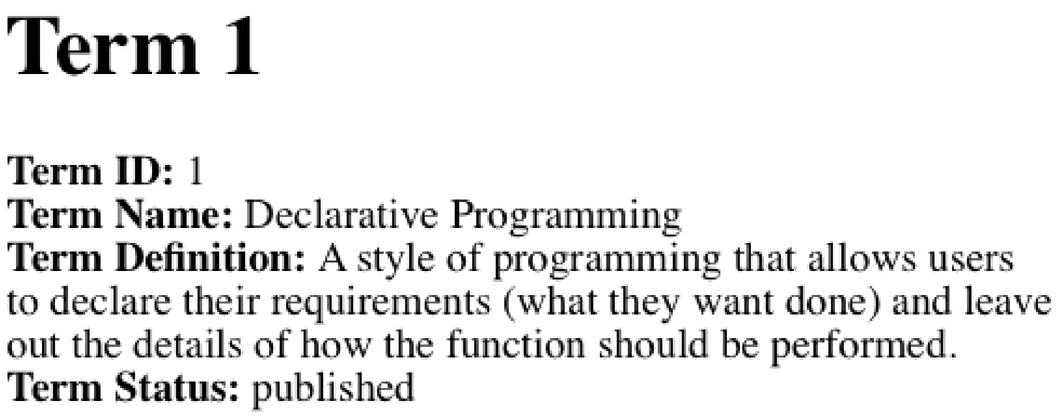
Our next step is to get the parameter from the URL to select the correct item. This is done by using the function request:get-parameter(). We then display all the elements of the term using one element per line. Example 1.3, “/db/apps/terms/views/view-item.xq” shows what the source of the `list-item.xq` file looks like

Example 1.3. /db/apps/terms/views/view-item.xq

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";

let $id := request:get-parameter("id", "")
let $term := collection('/db/apps/terms/data')/term[id=$id]

return
<html>
   <head>
      <title>Term {$term/id/text()}</title>
    </head>
    <body>
       <h1>Term {$term/id/text()}</h1>
       <b>Term ID: </b> {$term/id/text()}<br/>
       <b>Term Name: </b> {$term/term-name/text()}<br/>
       <b>Term Definition: </b> {$term/definition/text()}<br/>
       <b>Term Status: </b> {$term/publish-status-code/text()}<br/>
    </body>
</html>
```

This produces the output shown in Figure 1.2, “Output from `view-item.xq`”:

Figure 1.2. Output from `view-item.xq`



Term 1

Term ID: 1
Term Name: Declarative Programming
Term Definition: A style of programming that allows users to declare their requirements (what they want done) and leave out the details of how the function should be performed.
Term Status: published

Searching Items

There are three items that we must create to create a searchable application. These are the HTML search form, a search service and the configuration file for defining the indexes. In addition to these three files we have also provide a script that runs a reindex of the collection of terms.

Search Configuration File

Example 1.4, “`/db/system/config/db/apps/term/data/collection.xconf`” is a search configuration file that is stored in the collection `/db/system/config/db/apps/term/data`.

Example 1.4. `/db/system/config/db/apps/term/data/collection.xconf`

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
    <index>
        <!-- Disable the standard full text index -->
        <fulltext default="none" attributes="no"/>

        <!-- Range index configuration on the id attribute -->
        <!-- Most ids are integers but we will keep this general <create qname=
        <create qname="id" type="xs:string"/>

        <!-- Lucene index configuration -->

        <lucene>
            <!-- Use the standard analyzer will ignore stopwords like 'the', 'a
            <analyzer class="org.apache.lucene.analysis.standard.StandardAnalyz

            <!-- an index boost can be used to give matches in the
            name a higher score. This means a name match will have higher rank
            an match in the definition. -->
            <text match="//term/term-name" boost="2"/>
            <text match="//term/definition"/>
            <text match="//term/publish-status-code"/>
        </lucene>
    </index>
</collection>
```

This configuration file creates an index for the term id for fast searching. It also creates a Lucene fulltext index for all elements in the term.

Reindexing

After you have created or modified your configuration file you must reindex any data that you have. This can be done by pasting the following two lines into the XQuery sandbox:

Paste the following lines into the XQuery Sandbox to reindex the collection:

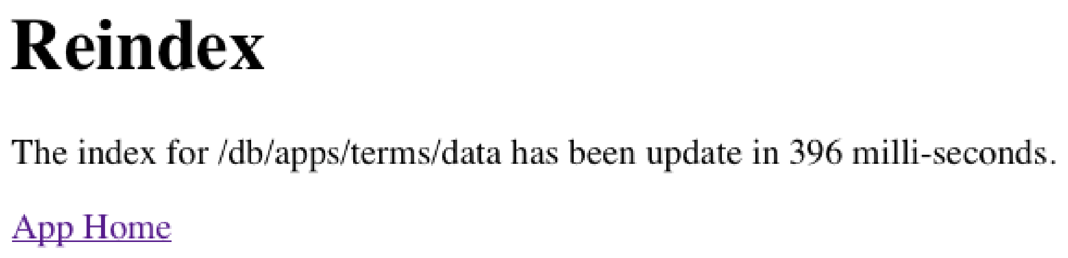
```
(xmldb:login('/db/apps/terms/data', 'admin', 'myadminpassword'),
 xmldb:reindex('/db/apps/terms/data'))
```

If you are not familiar with the eXist sandbox you can also run the XQuery script in Example 1.5, “`/db/apps/terms/admin/reindex.xq`”, which will display the results shown in Figure 1.3, “Results of running `reindex.xq`”. It is stored under the admin collection. The script will login as the administrator and then run the reindex function on the collection. It also returns the time it took to reindex the collection. For collections that are under 1,000 medium sized 10K byte documents this is usually runs in a few seconds. Tools are available for larger collections to schedule indexing during off hours with the eXist job scheduler.

Example 1.5. `/db/apps/terms/admin/reindex.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";

let $data-collection := '/db/apps/terms/data'

let $login := xmldb:login($data-collection, 'admin', 'myadminpassword')
let $start-time := util:system-time()
let $reindex := xmldb:reindex($data-collection)
let $runtime-ms := ((util:system-time() - $start-time) div xs:dayTimeDuration('

return
<html>
    <head>
       <title>Reindex</title>
    </head>
    <body>
    <h1>Reindex</h1>
    <p>The index for {$data-collection} has been update in {$runtime-ms} milli-
    <a href="../index.html">App Home</a>
    </body>
</html>
```

Figure 1.3. Results of running `reindex.xq`



Reindex

The index for /db/apps/terms/data has been update in 396 milli-seconds.

App Home

The Search Form

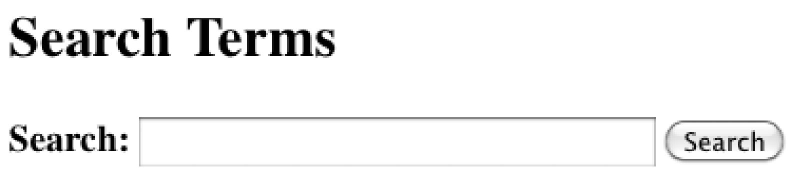
The search form is a simple HTML GET form with one text field input and one submit button. The action of this form (see Example 1.6, "`/db/apps/terms/search/search-form.html`") will use the value in the input field and send the field in the *q* parameter to the search service.

Example 1.6. `/db/apps/terms/search/search-form.html`

```
<html>
    <head>
        <title>Search Terms</title>
    </head>
    <body>
        <h2>Search Terms</h2>
        <form method="GET" action="search.xq">
            <b>Search:</b>
            <input name="q" type="text" value="" size="30"/>
            <input type="submit" value="Search"/>
        </form>
    </body>
</html>
```

Figure 1.4, "Blank search screen form" is a screen image of a blank search screen form. To use the search the user simply enters one or more keywords into the search form selects the search button (or enter key).

Figure 1.4. Blank search screen form



If you put a keyword in the input field the following URL will get generated:

```
/db/apps/terms/search/search.xq?q=mykeyword
```

The Search Service

The search service is an XQuery script that calls the Lucene fulltext search function.

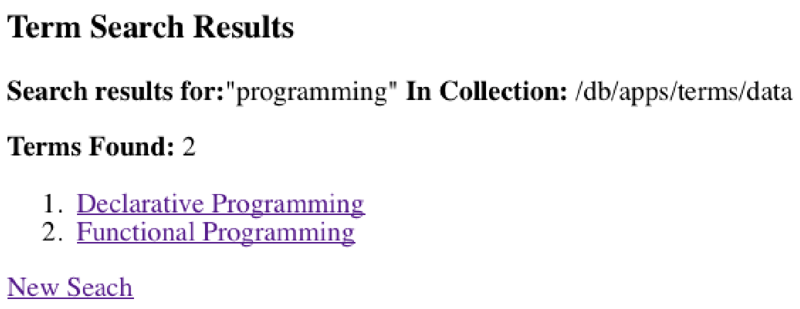
Example 1.7. `/db/apps/terms/search/search.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";


let $data-collection := '/db/apps/terms/data'
let $q := request:get-parameter('q', "")

(: put the search results into memory using the eXist any keyword ampersand equa
let $search-results := collection($data-collection)/term[ft:query(*, $q)]
let $count := count($search-results)

return
<html>
    <head>
       <title>Term Search Results</title>
     </head>
     <body>
        <h3>Term Search Results</h3>
        <p><b>Search results for:</b>&quot;{$q}&quot; <b> In Collection: </b>{$
        <p><b>Terms Found: </b>{$count}</p>
     <ol>{
           for $term in $search-results
              let $id := $term/id
              let $term-name := $term/term-name/text()
              order by upper-case($term-name)
          return
            <li>
               <a href="../views/view-item.xq?id={$id}">{$term-name}</a>
            </li>
      }</ol>
      <a href="search-form.html">New Search</a>
      <a href="../index.html">App Home</a>
   </body>
</html>
```

Search Results

The form will then pass the search keywords to the search service. The search service will return a series of search results with one line per hit. Each entry is also a link to the item-viewer service (see Figure 1.5, “Search results”).

Figure 1.5. Search results



Term Search Results

Search results for:"programming" **In Collection:** /db/apps/terms/data

Terms Found: 2

1. Declarative Programming
2. Functional Programming

New Seach

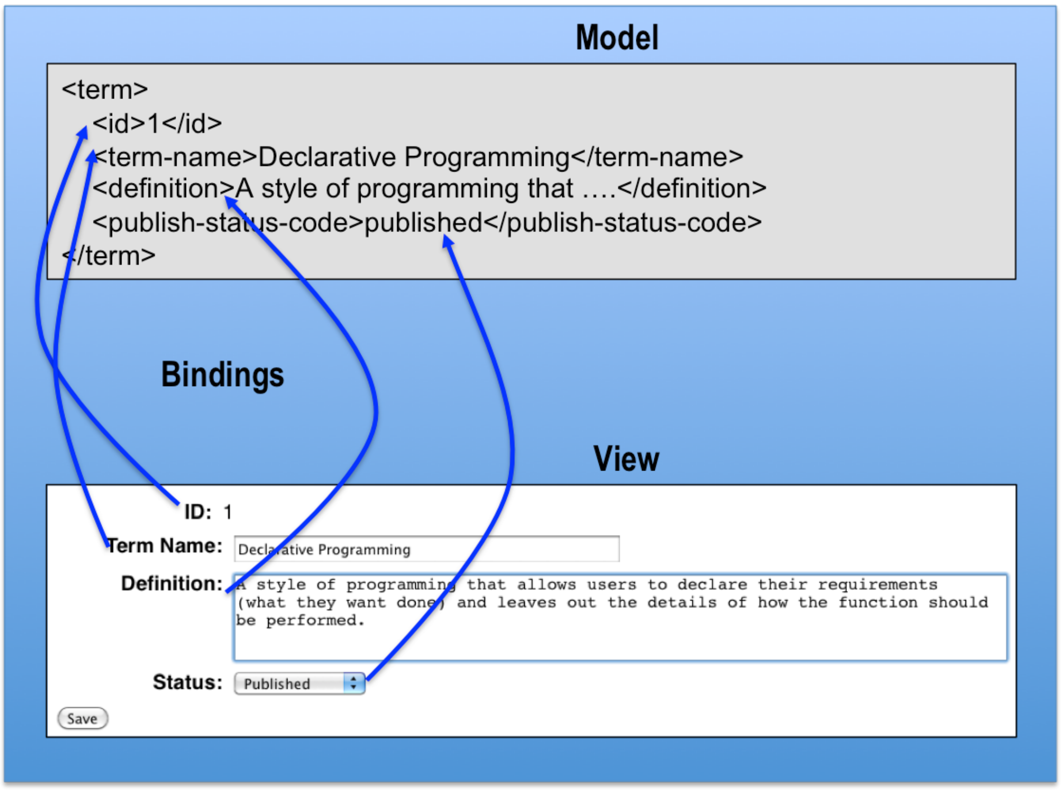
Editing

Editing the items is the most complex portion of building an XRX application. Because of this many advanced XRX frameworks attempt to automate this process by generating all of the required files

Model-View-Binding

To understand how the `edit.xq` script works it is first important to understand how the XForms standard uses Model-View-Binding to associate a user interface control with an XML instance inside the model. This is illustrated in Figure 1.6, "Model View Binding".

Figure 1.6. Model View Binding



```
<term>
   <id>1</id>
   <term-name>Declarative Programming</term-name>
   <definition>A style of programming that ….</definition>
   <publish-status-code>published</publish-status-code>
</term>
```

Model
Bindings
View
ID: 1
Term Name: Declarative Programming
Definition: A style of programming that allows users to declare their requirements (what they want done) and leaves out the details of how the function should be performed.
Status: Published
Save

In the form the XML data that the form modifies is loaded into an `<xs:instance>` element within the `<xf:model>`. This is specified using the `src` attribute. Inside the body of the form each of the user interface controls (an output, input, textarea and select1 control) each have a `ref` attribute. This attribute contains the XPath expression of the element it corresponds to within the model.

The Edit Query

The Edit query is the most complex file in this application. It must perform saves for new items as well as update operations. The source code in Example 1.8, "`/db/apps/terms/edit/edit.xq`" should be studied carefully since many of the techniques used in the form will be used in more complex forms.

Example 1.8. `/db/apps/terms/edit/edit.xq`

```
                       /* align the labels but not the save label */
                        xf|output xf|label, xf|input xf|label, xf|textarea xf|label, xf|sel
                             display: inline-block;
                             width: 14ex;
                             text-align: right;
                             vertical-align: top;
                             margin-right: 1ex;
                             font-weight: bold;
                         }

xf|input, xf|select1, xf|textarea, xf|ouptut {
       display: block;
       margin: 1ex;
  }
]]>
       </style>
            <xf:model>
                <xf:instance xmlns="" src="{$file}" id="save-data"/>
                <xf:submission id="save" method="post" action="{if ($new='true'
            </xf:model>
            </head>
            <body>
                <h1>Edit Term</h1>

                {if ($id)
                  then (
                    <xf:output ref="id" class="id">
                        <xf:label>ID:</xf:label>
                    </xf:output>
                ) else ()}

                <xf:input ref="term-name" class="term-name">
                    <xf:label>Term Name:</xf:label>
                </xf:input>

                <xf:textarea ref="definition" class="definition">
                    <xf:label>Definition:</xf:label>
                </xf:textarea>

                <xf:select1 ref="publish-status-code">
                    <xf:label>Status:</xf:label>
                    <xf:item>
                        <xf:label>Draft</xf:label>
                        <xf:value>draft</xf:value>
                    </xf:item>
                    <xf:item>
                        <xf:label>Under Review</xf:label>
                        <xf:value>review</xf:value>
                    </xf:item>
                    <xf:item>
                        <xf:label>Published</xf:label>
                        <xf:value>published</xf:value>
                    </xf:item>
                </xf:select1>

                <xf:submit submission="save">
                    <xf:label>Save</xf:label>
                </xf:submit>
            </body>
            </html>

            let $xslt-pi := processing-instruction xml-stylesheet {'type="text/

            return ($xslt-pi, $form)
```

One item to note is that this form does "double duty" as both a form for new items as well as a form for updating existing items. The *`new=true`* parameter must always be passed to the form when creating a new item. Production systems check for these parameters and return error codes if one or the other is not passed to the form.

All XForms store the data that is manipulated in the tag `<xf:model>`. This form uses a single instance within the model to store the data that will saved when the users selects the "Save" button. The save button in XForms is called the `<xf:submit>` element. It has a single attribute called the submission attribute that is associated with an `<xf:submission>` element within the model. In our example above the name of the submission element (its id) is `save`. The save submission element is responsible for sending the data from the XForms client to a specific service on the server. In the example above there are two slightly different XQuery services, one for saving new items and one for updating existing items. We will be covering the save-new and the update queries later in this tutorial.

The query that is used is wrapped inside of the action attribute of the save submission. Here is that code:

```
action="{if ($new='true')
            then ('save-new.xq')
            else ('update.xq')}"
```

You can see that if the user is creating a new item the data is sent via an HTTP POST to the `save-new.xq` script. If the user does not have a new item the POST data is sent to the `update.xq` script.

Although we could have used a single save.xq script this structure allows you to trigger different behavior for different functions you may want. For example the save-new.xq might also trigger an e-mail notification when new records are saved for the first time. Versioning might be triggered only when the file is updated. Advanced user guides will have examples of both of these functions.

The next section of code to notice is that the ID element is only displayed using a read-only `<xf:output>` tag if the form is in update mode.

```
{if ($id)
      then (
        <xf:output ref="id" class="id">
            <xf:label>ID:</xf:label>
        </xf:output>
    ) else ()}
```

This shows some of the power of combining XQuery and XForms. In this case we are using logic on the server to conditionally include portions of the form based on the context. The process of using context such as mode, user, group, role and project is central to understanding how forms can be dynamically created to precisely meet the needs of your users. No more "one size fits all". No more forcing users to fill out field of forms that are not relevant to their situation. XRX forms can all be dynamically created directly as they are needed. We can use both client and server logic to determine what features of the form are enabled. XForms includes function called `<xf:bind>` that also uses XPath expressions to determine if fields should be displayed. This will also be covered in advanced tutorials.

The next item to note is that there are four different user interface controls in this form. The first one is a read-only output. The second is the `<xf:input>` control that gathers input in a single line. The third is a `<xf:textarea>` control that allows users to enter multi-line descriptions for definitions of terms. The last control is the `<xf:select1>` control that allows the user to select one value from a list of values. For a complete discussion of the XForms controls we suggest you use the XForms Wikibook at http://en.wikibooks.org/wiki/XForms. The Input Form Controls section goes through each of the controls in the XForms specification. In addition to the standard controls there are other controls that can also be integrated directly into XForms such as rich-text editors.

Each of the input controls has a ref attribute that indicates what element in the instance it is bound to. If you have multiple instances and multiple models you many not be able to use all the default values like in this example. This ref attribute is how leaf elements the model get bound to each input control. In general, when you are building simple flat forms there is a one-to-one correspondence between the form elements and the instances in the model. Complex forms also allow you to have repeating elements so you can add one-to-many structures in a form. This means that XForms are not restricted to managing flat list of elements. They can contain multiple nested elements with elements. This will also be discussed in advanced tutorials.

The final part of the form (see Example 1.9, “XSLTForms processing instruction” contains the instructions needed to place the XSLTForms processing instruction at the top of the file when it is rendered.

Example 1.9. XSLTForms processing instruction

```
let $xslt-pi := processing-instruction xml-stylesheet
                {'type="text/xsl" href="/exist/rest/db/xforms/xsltforms/xsltfor
return ($xslt-pi, $form)
```

You can also add a directive that will put the XSLTForms system into a debug mode by adding the following.

```
let $debug := processing-instruction xsltforms-options {'debug="yes"'}
return ($xslt-pi, $debug, $form)
```

Saving New Items

The save new process must first access the XML file that stores the next ID to be used to create a unique file name. We store the next ID to be used in a small XML file with only one element in the root called `next-id` (see Example 1.10, “`/db/apps/terms/edit/next-id.xml`”

Example 1.10. `/db/apps/terms/edit/next-id.xml`

```
<data>
    <next-id>6</next-id>
</data>
```

The `<next-id>` element is updated using an XQuery "update function" when new items are saved to the data collection. In this case we save the file using the number as a file name so the next file saved will be 6.xml. After the file is saved the number is incremented so that the `next-id` will be 7. This is similar to the auto-increment function in many other databases. eXist also has a counter function that you can use. Using an arbitrary number as an ID is sometimes called a foreign key since it is external to the actual data in the XML file.

When you create files, sometimes you want to create an identifier that is not just a number but it might also serve some meaning to allow users to differentiate items in a collection. For example a database of countries might use a country name as the file name. You can also allow users to pick an identifier and check for duplicates as they enter the data (see Example 1.11, “`/db/apps/terms/edit/save-new.xq`”). This will be covered in advanced sections.

Example 1.11. `/db/apps/terms/edit/save-new.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";

(: save-new.xq :)

let $app-collection := '/db/apps/terms'
let $data-collection := '/db/apps/terms/data'

(: this is where the form "POSTS" documents to this XQuery using the POST metho
let $item := request:get-data()

(: get the next ID from the next-id.xml file :)
let $next-id-file-path := concat($app-collection,'/edit/next-id.xml')
let $id := doc($next-id-file-path)/data/next-id/text()
let $file := concat($id, '.xml')

(: this logs you into the collection :)
let $login := xmldb:login($app-collection, 'admin', 'myadminpassword')

(: this creates the new file with a still-empty id element :)
let $store := xmldb:store($data-collection, $file, $item)

(: this adds the correct ID to the new document we just saved :)
let $update-id :=  update replace doc(concat($data-collection, '/', $file))/ter

(: this updates the next-id.xml file :)
let $new-next-id :=  update replace doc($next-id-file-path)/data/next-id/text()

(: we need to return the original ID number in our results, but $id has already
let $original-id := ($id - 1)

return
<html>
    <head>
       <title>Save Conformation</title>
    </head>
    <body>
    <a href="../index.xhtml">Term Home</a>
    <p>Term {$original-id} has been saved.</p>
    <a href="../views/list-items.xq">List all Terms</a>
    </body>
</html>
```

Updating Existing Items

The update function is simpler then the save function since it does not have to worry about creating a new file and incrementing a counter. It simply takes the incoming POST data and stores it in the file. Note that by default this means that the entire data file is updated and reindexed upon the store operation. eXist does contain versioning and it can be enabled by simply configuring a single XML file in the `/db/system/config` area (see Example 1.12, “ `/db/apps/terms/edit/update.xq` ”.

Example 1.12. `/db/apps/terms/edit/update.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";

let $title := 'Update Confirmation'
let $data-collection := '/db/apps/terms/data'

(: this is where the form "POSTS" documents to this XQuery using the POST metho
let $item := request:get-data()

(: this logs you into the collection :)
let $login := xmldb:login($data-collection, 'admin', 'myadminpassword')

(: get the id out of the posted document :)
let $id := $item/id/text()

let $file := concat($id, '.xml')

(: this saves the new file and overwrites the old one :)
let $store := xmldb:store($data-collection, $file, $item)

return
<html>
    <head>
       <title>{$title}</title>
    </head>
    <body>
    <h1>{$title}</h1>
    <p>Item {$id} has been updated.</p>
    </body>
</html>
```

Deleting

Deleting items is much simpler then editing items. There are only two files that we will need to create. Each of them take a single REST parameter. The first file is a confirmation XQuery script that just asks the user “Are you sure you want to delete this term?”. The second script actually does the deletion.

Confirming Delete

The delete confirmation script takes the ID of the item to be deleted and opens the document using the doc() function. It then presents the user with details about the item and displays two choices. One to delete and the other to cancel the delete. A CSS file can be used to color the links appropriately with a red warning indicator. See Example 1.13, “`/db/apps/terms/edit/delete-confirm.xq`”.

Example 1.13. `/db/apps/terms/edit/delete-confirm.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";

let $id := request:get-parameter("id", "")

let $data-collection := '/db/apps/terms/data/'
let $doc := concat($data-collection, $id, '.xml')

 return
<html>
    <head>
        <title>Delete Confirmation</title>
        <style>
  <![CDATA[
   .warn {background-color: silver; color: black; font-size: 16pt; line-height:
  ]]>
    </style>
    </head>
    <body>
        <a href="../index.xhtml">Item Home</a> &gt; <a href="../views/list-item
        <h1>Are you sure you want to delete this term?</h1>
        <b>Name: </b>{doc($doc)/term/term-name/text()}<br/>
        <b>Path: </b> {$doc}
        <br/>
        <br/>
        <a class="warn" href="delete.xq?id={$id}">Yes - Delete This Term</a>
        <br/><br/>
        <br/>
        <a  class="warn" href="../views/view-item.xq?id={$id}">Cancel (Back to
    </body>
</html>
```

The Delete Script

The delete script (see Example 1.14, “ `/db/apps/terms/edit/delete.xq` ”) also takes a single REST parameter of the ID

Example 1.14. `/db/apps/terms/edit/delete.xq`

```
xquery version "1.0";
declare option exist:serialize "method=xhtml media-type=text/html indent=yes";

let $data-collection := '/db/apps/terms/data'

(: this script takes the integer value of the id parameter passed via get :)
let $id := request:get-parameter('id', '')

(: this logs you into the collection :)
let $login := xmldb:login($data-collection, 'admin', 'myadminpassword')

(: this constructs the filename from the id :)
let $file := concat($id, '.xml')

(: this deletes the file :)
let $store := xmldb:remove($data-collection, $file)

return
<html>
    <head>
        <title>Delete Term</title>
        <style>
<![CDATA[
   .warn  {background-color: silver; color: black; font-size: 16pt; line-height
]]>
  </style>
  </head>
                <body>
                    <a href="../index.xhtml">Terms Home</a> &gt; <a href="../vi

                    <h1>Term id {$id} has been removed.</h1>
                </body>
                </html>
```

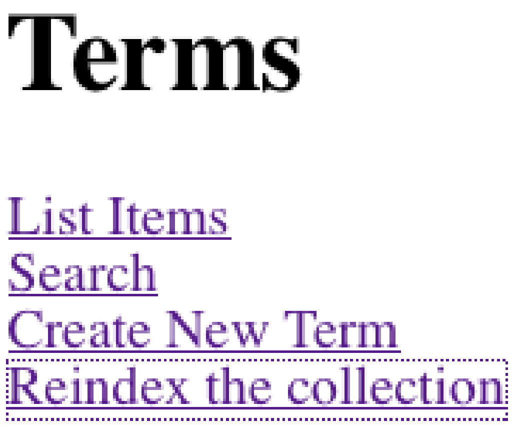
The Application Home Page

A simple application home page can be a description of the application and a static list of links to the main entry points of the application: the item lister, the search form, the create new and the reindex. The index page can be a static HTML page as described in Example 1.15, “`/db/apps/terms/index.html`”. See Figure 1.7, “Output of `index.html`” for the output.

Example 1.15. `/db/apps/terms/index.html`

```
<html>
    <head>
        <title>Terms</title>
    </head>
    <body>
        <h1>Terms</h1>
        <a href="views/list-items.xq">List Items</a>
        <br/>
        <a href="search/search-form.html">Search</a>
        <br/>
        <a href="edit/edit.xq?new=true">Create New Term</a>
        <br/>
        <a href="admin/reindex.xq">Reindex the collection</a>
        <br/>
    </body>
</html>
```

Figure 1.7. Output of `index.html`



Terms

List Items
Search
Create New Term
Reindex the collection

Next Steps

If you have managed to learn all of the CRUDS functions you are now ready to move on the some more complex examples. Here are some suggestions for next steps.

1. Create a collection `/db/apps/modules` and add a file called `style.xml` in that collection. Add XQuery functions for `style:header()`, `style:footer()` and then reference these functions in each of your HTML web pages.

2. Change the `list-items.xq` to use HTML tables to view each item.

3. Use the eXist permission system to create a groups called “editor” and a group called “term-admin”. Change the group permissions on the edit and admin collections to only allow users in these groups to be able to access these collections.

4. In the `list-items.xq` query, use XQuery sequences to pre-sort items and then display only an initial subset of the data using the subsequence function.

5. Add URL parameters *`start`* and *`num`* to the list-items to indicate what record to start to display and how many records to display.

6. Learn how to create one-to-many relationships in your forms using the `<xf:repeat>` element. For example create a form that allows you to add multiple phone numbers to a contact record or multiple authors to a book entry.
7. Learn how to use XForms binding to conditionally display elements in a form.
8. Get fancy with how tables of data are displayed. Add sorting to table columns.
9. Add security to your forms by only allowing people in an "edit" group to be able to write to the data collection.

   Create roles for users such as editor, publisher and then copy the XML files to a remote host using the `http-client()` functions.
10. Add forms that edit complex data using in-browser lists and inspectors.
11. Create forms that manage document workflows. Add workflow steps that flow to the right as they expand.
12. Create advanced search forms that use multiple selection criteria such as document types, authors or date ranges.
13. Create complex business logic in how selection lists can be controlled. Use one selection list to control the values of a second selection list.
14. Move all of the codes in the publish-status selection list into an XML file and place it in a collection called `code-tables`. Then add an instance to the form that reads this code table into the form.
15. Add a XQuery function that will take a status codes value and return its label.
16. Modify the system configuration file for the `/db/apps/terms/data` collection to enable versioning when items are updated.

References

All of these topics and many more are covered in the XQuery, XForms and XRX Wikibooks. You can use the search tools within Wikibooks to find how specific elements are used within each of the examples.

Wiki Books

[xforms-wiki] *XForms Wikibook*. http://en.wikibooks.org/wiki/XForms. Dan McCreary.

[xrx-wiki] *XRX Wikibook*. http://en.wikibooks.org/wiki/XRX. Dan McCreary.

[xquery-wiki] *XQuery Wikibook*. http://en.wikibooks.org/wiki/XQuery. Chris Wallace. Dan McCreary.

[dubinko] *XForms Essentials*. Micah Dubinko. O'Reilly Publishing. Copyright © 2003.

Articles

[wikipedia-xrx] *XRX (web application architecture)*. http://en.wikipedia.org/wiki/XRX_%28web_application_architecture%29.

[tennison-xrx-in-exist] *XRX: XQueries in eXist*. Jeni Tennison. http://news.oreilly.com/2008/07/xrx-xqueries-in-exist.html.

[simple_elegant_disruptive] *XRX: Simple, Elegant, Disruptive*. Dan McCreary. http://www.oreillynet.com/xml/blog/2008/05/xrx_a_simple_elegant_disruptiv_1.html.

Books

[walmsley] *XQuery*. Priscilla Walmsley. O'Reilly Publishing. Copyright © 2007.

[dubinko] *XForms Essentials*. Micah Dubinko. O'Reilly Publishing. Copyright © 2003.