XQuery & eXist-db

Scripts and Manuscripts, May 8, 2025

Jean-Paul Rehr, Università di Torino / UMR 5468 CIHAM

Objectives

- Orientation to eXist-db
- Introduction to Xquery: a declarative, functional language
- Thinking with XQuery
- Accessing and manipulating data
- Output 1: simple dynamic page
- Output 2: page with table of contents, linked to “edition”
- Output 3: dynamically generate “edition” of each deposition

Orientation to eXist-db

- Navigating the eXist-db ecosystem: a tour with essential vocabulary
- Configuring eXist-db - why? how?
- Accessing eXist-db - native development tools

The eXist-db ecosystem

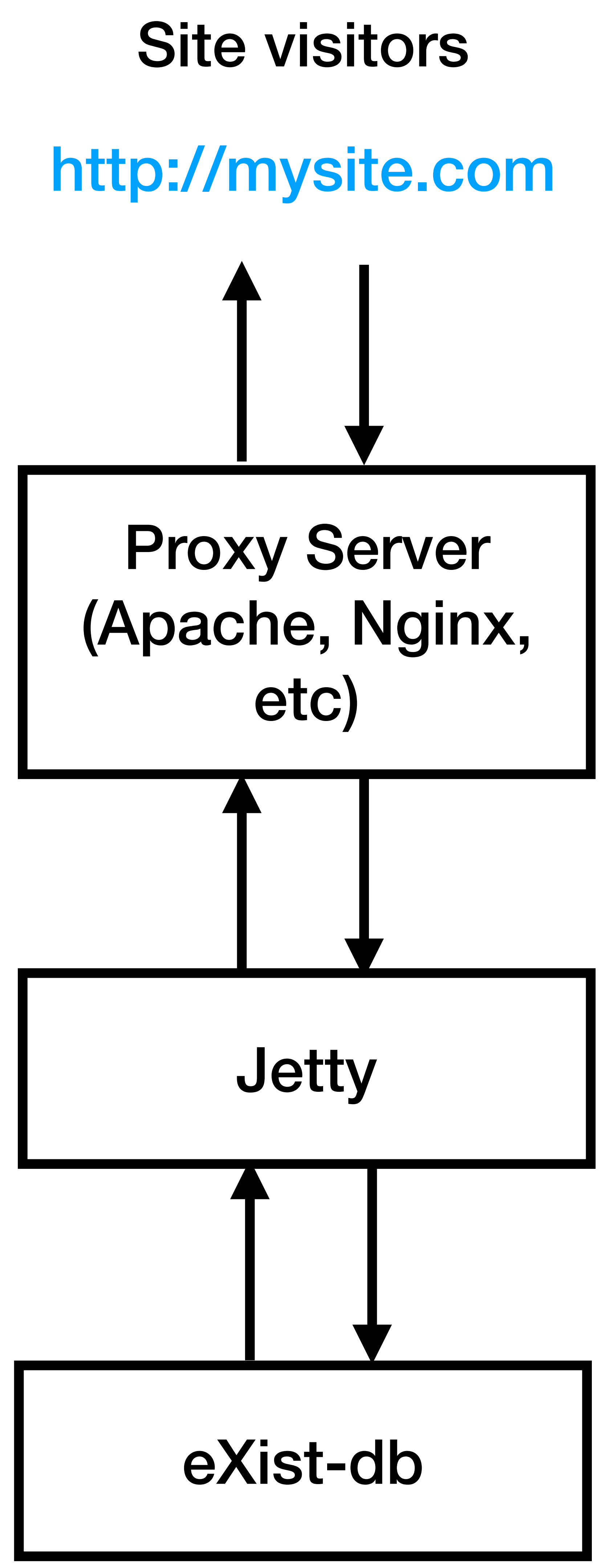
- A “NoSQL” database: XML-centric database and application server supports:
  - XML, (X)HTML, JSON and binary documents (PDF, JPG, etc)
  - Complete application development
  - “Pipeline” management:
    - XPath, XQuery, XSLT, XSL-FO
    - Support through XQuery 3.1, XSLT 3.0
    - Built-in development tools

eXist-db: structure

For web servers

Server development access:

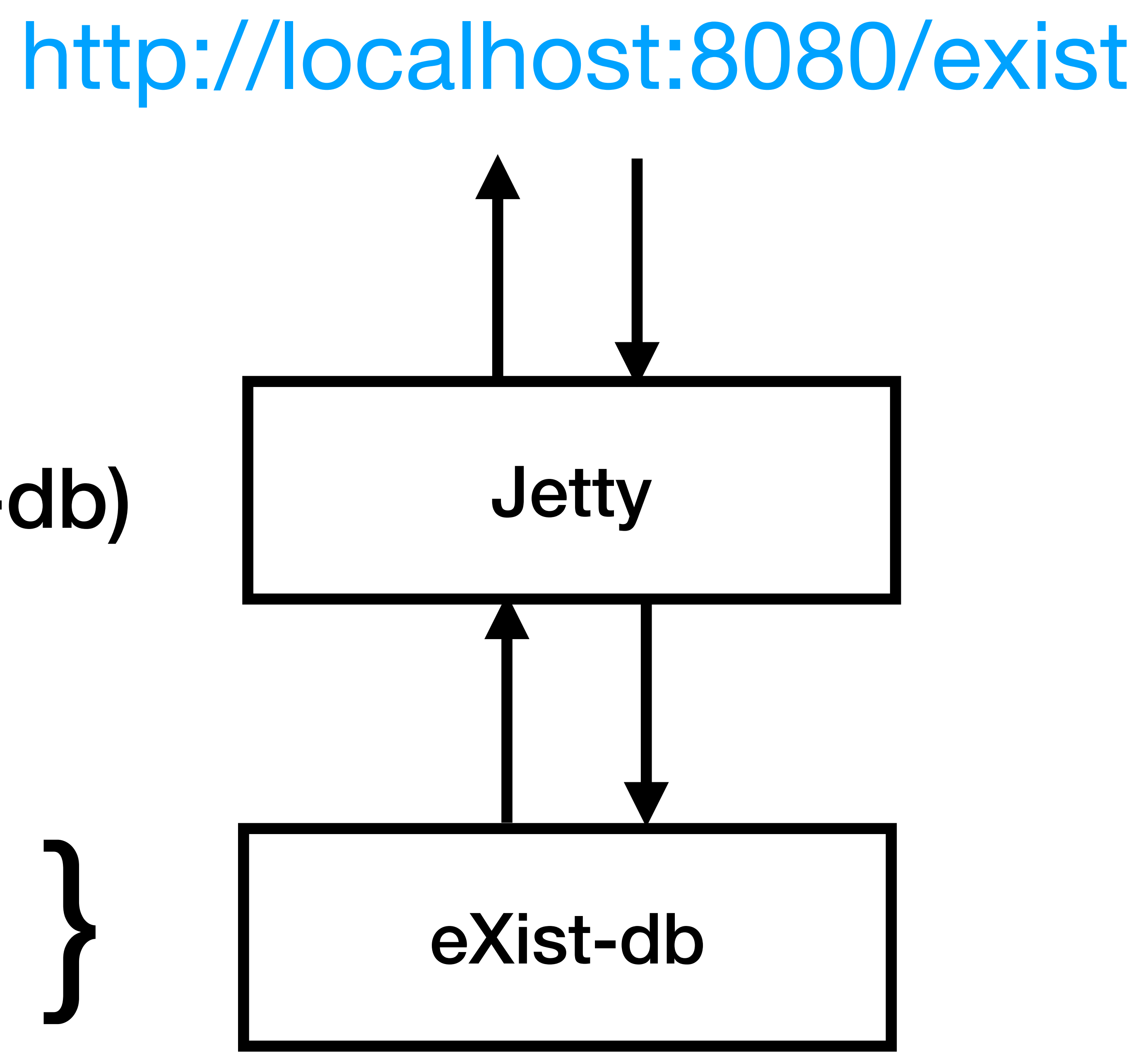
- database: http://mysite.com/exist
- configuration: via “SSH”



eXist-db: structure

On localhost

Java web application server (installs automatically with eXist-db)



$EXIST_HOME/bin —> scripts (start, stop, etc)

$EXIST_HOME/etc —> configuration files

$EXIST_HOME/logs —> activity logging

}

Configuring eXist-db

To manage database functionality (Java plug-ins, services, port access)

- How?
    - “`$EXIST_HOME`" indicates the filesystem directory of your eXist installation
    - `$EXIST_HOME/bin` —> scripts (start, stop, etc)
    - `$EXIST_HOME/etc` —> configuration files
    - `$EXIST_HOME/logs` —> activity logging
- Configuration changes do not take effect until restart!

eXist-db: types of deployment

- database
- applications (“packages”):
    - eXist-db templating
    - roll-your-own (custom) application

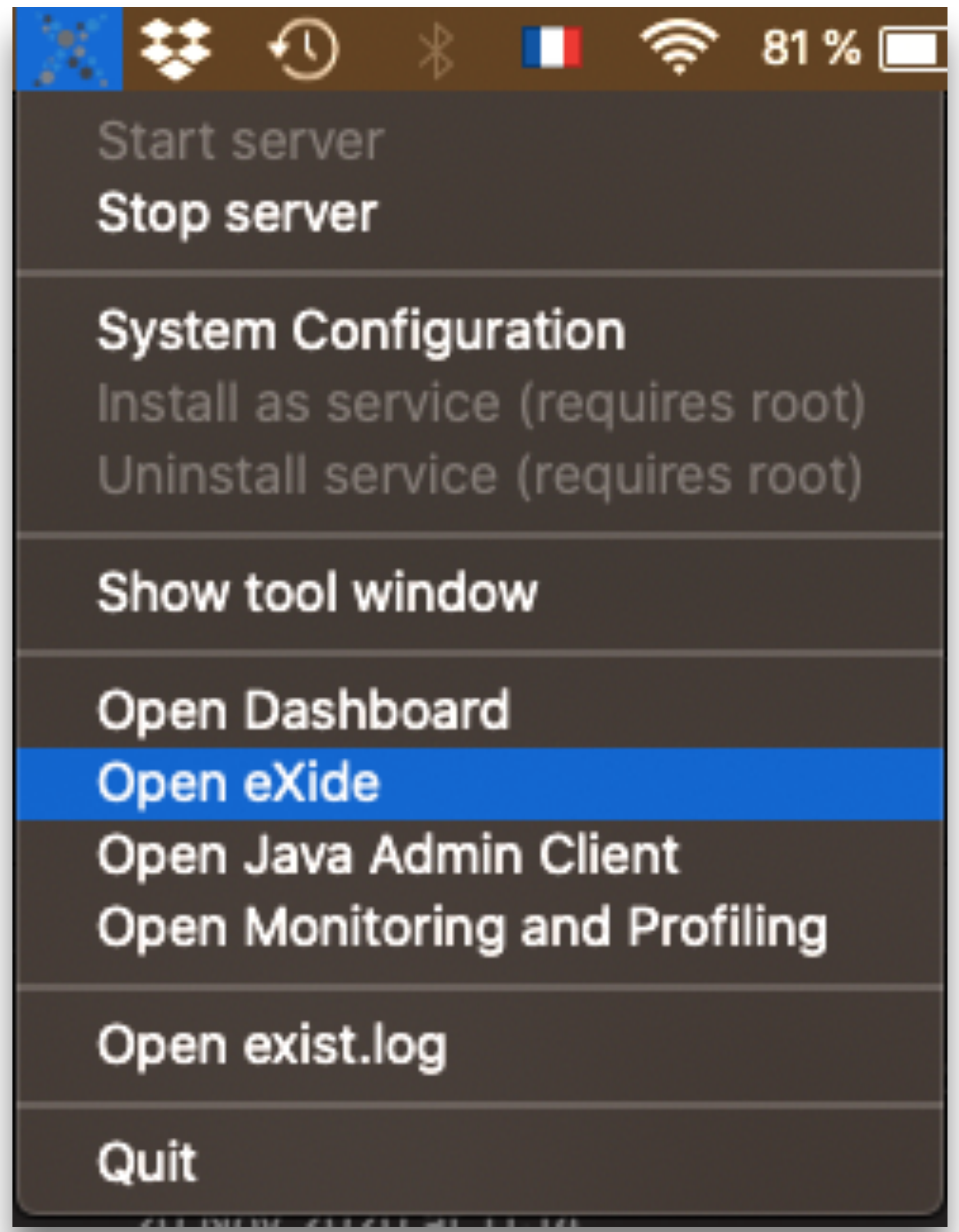
As a schema-less database, all objects are treated the same - the “application” environment is a layer of additional tools provided by eXist in the form of “packages". Those tools are just other objects in the database (.xml, .json, .xql, etc)

eXist-db: database access methods

- HTTP:
  - direct object requests, for example:
    - http://localhost:8080/exist/apps/<appname>/data.xml
    - http://localhost:8080/exist/apps/<appname>/myquery.xql
  - via package/application controller (controller.xql)
    - http://localhost:8080/exist/apps/<appname>
  - REST: http://localhost:8080/exist/apps/<appname> (with REST annotation)

Accessing the database

- In the “system tray”:



- dashboard: http://localhost:8080/exist/apps/dashboard (note your port!)
- Java admin client: in your system tray or run `$EXIST_HOME/bin/client.sh` (MacOS/Linux) or `$EXIST_HOME/bin/client.bat` (Win)

eXist-db: a typical application

- root (controller, html): `/db/apps/<appname>`
    - templates: `/db/apps/<appname>/templates`
    - modules: `/db/apps/<appname>/modules`
    - resources (assets): `/db/apps/<appname>/resources`

    - my own habits:
        - xslt files: `/db/apps/<appname>/xslt`
        - xml data: `/db/apps/<appname>/data`

eXist-db: how to make an application?

- a special “package”
  - .xar file
  - built using EXPath package model (https://expath.org/modules/pkg/)
- compile a base package using Yeoman generator
  - https://github.com/eXist-db/generator-exist
- Install via eXist dashboard

Why XQuery?

- “XQuery is domain-appropriate for digital humanists. XQuery allows for full-stack application development. XQuery is compact, terse, and relatively easy to learn.”
- Shared specifications with XPATH (overlapping XSLT)
- When you write XQuery:
  - you use XQuery **FLOWR statements** and **XPATH functions**
  - you can use (“embed”) XSLT and XSL-FO
  - manipulate JSON (`arrays` and `maps`)
  - read and write XML and (X)HTML, `serialize` to other formats
  - the code touches data, architecture, and environment

XQuery: quick theory

- “It is indeed tough for experienced programmers to switch from imperative, object-oriented programming to functional programming”
- Declarative, functional programming (e.g. XQuery):
    - declare functions, reuse them everywhere
  - In XQuery, every function has a clear input and output:
    - no secondary effects
    - variables are immutable

Remember your XQuery/XPATH functions!

Priscilla Walmsley’s functions:

https://www.datypic.com/xq/

David Birnbaum’s “The XPath functions we use most”:

http://dh.obdurodon.org/functions.xhtml

Path functions by example

https://maxtoroq.github.io/xpath-ref/

Tutorials…

Michael Kay* tutorials (with downloadable file):

Xquery in 10 minutes: http://www.stylusstudio.com/xquery-primer.html

FLOWR: http://www.stylusstudio.com/xquery-flwor.html

About Xquery as functions: http://www.stylusstudio.com/xquery/xquery-functions.html

*Michael Kay is one of the “inventors” of XSLT/Xquery standards, and is the creator of the company “Saxon” (if you use Oxygen, you will know the importance of Saxon!)

XQuery Basics

- These “statements” are all valid queries:
    - 1
    - "x"
    - ( )
    - 1+1
    - a=b
    - ("a","b","c")
- valid XPath are also valid queries, for example:
    - fn:count(("a","b","c"))
    - fn:max((1,2,3))
    - fn:replace("#MYID", "#", "")
    - fn:concat("Hello", " ", "world", "!")
    - fn:distinct-values(("a","b","a", "c", "a"))
    - fn:sort(("z","f","t"))

Get your data

- Organized as *collections* and *resources*
- *Resources* are XML, JSON, Binary
  - Get XML document: `fn:doc($path_to_resource)`
  - Get JSON document: `fn:json-doc($path_to_resource)`
  - Get Binary document: `util:binary-doc($path_to_resource)`
- This returns a collection of XML documents:
  - `collection($path_to_collection)`

Get some data!

```
xquery version "3.1";

doc("/db/apps/myapp/data/BMTOULOUSE-MS609/de_manso_sanctarum_puellarum/MS609-0001.xml")
```

This is XPath/Xquery…how do we get data from the document?

When we can access the data, how does it appear?

Get some data!

```
xquery version "3.1";

declare namespace tei="http://www.tei-c.org/ns/1.0";

doc("/db/apps/myapp/data/BMTOULOUSE-MS609/de_manso_sanctarum_puellarum/
MS609-0001.xml")//tei:body//tei:persName
```

The result is a `sequence`. **Every XQuery produces a `sequence`.**

Getting and manipulating a collection

Count the documents in a single collection

```
xquery version "3.1";

declare namespace tei="http://www.tei-c.org/ns/1.0";

count(collection("/db/apps/myapp/data/BMTOULOUSE-MS609/de_manso_sanctarum_puellarum"))
```

Getting and manipulating collections

FLOWR: let and **return** to count and add documents in a collection

Method 1: use Xquery **`let :=`** and **`return`**

```
xquery version "3.1";

declare namespace tei="http://www.tei-c.org/ns/1.0";

let $msp := count(collection("/db/apps/myapp/data/BMTOULOUSE-MS609/
de_manso_sanctarum_puellarum"))

let $sml := count(collection("/db/apps/myapp/data/BMTOULOUSE-MS609/
de_sancto_martino_lalanda"))

return $msp + $sml
```

What is an XPath-only solution to the above?

Objective 1

- Output count to a “home” page
    - Request comes into the app controller (/db/apps/<appname>/controller.xql)
    - The controller is a special eXist-only XQuery module
    - The controller “forwards” (sends) the request to a main module, redirects the request, obtains the requested resource
    - The main module needs to output HTML

XQuery: main and library modules

- A main module is not referencable (cannot be imported into other modules)
- A library module provides multiple named functions (can be imported into other modules)
- There is no standard file extension to distinguish them:
    - .xq, .xql, .xqm, .xquery, etc
- Be internally consistent with file naming! (I use .xql for libraries and .xqm for modules, others use the inverse, or .xq and .xqm)

Objective 2

- Output a page with an **HTML table** of the confessions found in **de_manso_sanctarum_puellarum**
- 2 columns
    - document ID (as link) (TEI/@xml:id)
    - deponent name (as text) (TEI//body//persName[@role = “dep”])
- Order them by deponent

FLOWR: for…return

Iterating through a sequence

- a “sequence” is a fundamental concept in XQuery (and XPATH)
- a sequence can be composed of
    - strings, numbers, dates, or other “atomic values”
        - `("a", "b", "c", "d", "e")`
    - nodes
    - and anything else!
- Why use `for`? To treat each *item* in a *sequence*

FLOWR: for…return

Iterating through a sequence

- the typical use of `for` takes the structure:

```
for $myvar in $mysequence
return $myvar
```

- the sequence can be expressed literally:
  - `for $myvar in (1, 2, 3, 4, 5)`
- or expressed through variables:
  - `for $myvar in $myvariable`

FLOWR: for…return

Iterating through a sequence

```
xquery version "3.1";

for $x in ("a", "b", "c", "d", "e")

return $x
```

```
xquery version "3.1";

for $x in 1 to 10

return $x
```

FLOWR: order of statements

- Despite the order suggested by the acronym FLOWR, we can use `let` anywhere. It is frequently used at the beginning to create variables used in the `for` statements:

```
let $myseries := ("a", "b", "c", "d", "e")

for $x in $myseries

return $x
```

FLOWR: order by

`order by` allows us to sort the results

- it must appear after the `let` statements used in `order by`
- it appears before `return`
- `order by` can take ascending/descending, and can be used in a hierarchy
  - `order by $myvar`
  - `order by $myvar ascending`
  - `order by $myvar2 descending, $myvar2 descending`

FLOWR: order by - warning!

```
xquery version "3.1";

for $x in ("1", "2", "3", "11", "22", "33")

order by $x

return $x
```

```
xquery version "3.1";

for $x in (1, 2, 3, 11, 22, 33)

order by $x

return $x
```

FLOWR: where

`where` allows us impose filters on the results

- appears before `return`
- `where` takes any XPATH statement

`where` versus XPATH predicates?

- we prefer XPATH predicates as they are generally more efficient with database indexes
- but sometimes the filter logic we want to apply can’t be achieved with predicates
- implementation-specific (testing)

XQuery: function arguments

- why? to allow functions to receive data/criteria from other functions (**arguments**) and to impose criteria for basic validation
  - `module:function($arg1 as type, $arg2 as type, etc) as type`
  - `test:count-people($document as node()) as xs:integer`

XQuery: function arguments

- function arguments take the form of
  - variables `$arg` with a `type`
    - `type` provides **constraint** and/or **validation**
    - `type*` indicates multiple items allowed (**sequence**)
    - `type?` indicates **optional**
    - **type** can be
      - `xs:string`, `xs:integer`, etc (xml datatypes); or
      - `item()`, `node()`, `map()`,etc

Objective 3

- Output an edition of each confession /tei:body dynamically based on the document ID
- Simple transformation using embedded XSLT, producing plain text in paragraphs (<p>) based on <tei:seg>, and with all person names and place names as links (<a>)
    - persName, placeName
    - the resource is persName/@ref, placeName/@ref
- Put the different page view (list/edition of text) into functions, use "if" to determine which function to call
- Move HTML into a library function

XQuery library function

To create a library function, we need at minimum (so that it can be “called” elsewhere!):

- `module namespace`
- `declare function`

Library function: module namespace

Declare module namespace:

- `module namespace test="http://exist-db.org/apps/modules/test"`
- every library module must have a unique namespace to be reference-able by other modules
- use consistent namespace construction in your app:
    - easier to remember, use, and **debug**
    - use characters: a-z, A-Z, 0-9, _, -
- the namespace does not need to be the same as the file name, this works equally:
- `module namespace foo="http://exist-db.org/apps/modules/test"`

Library function: “declaring” a function

Declare function:

- declaration, namespace:function-name
    - `declare function doc:doc-list`
- parameters
    - `($nodes as node()*, $username as xs:string?)`
- wrap the function body in
    - `{ };`

Library function: “calling” a function

A function is “called” through **two parts**:

- declare the namespace where we are using the function (import the module)
- name the function and fulfill the “arguments/parameters” (if there are any)