GETTING STARTED WITH TOPIC TOOLS

A brief tutorial.

Abstract

This tutorial gives you an impression how to develop topic tools, assisted by ttools. You will create the first version of a new topic tool which will offer operations for evaluating XML content. On the way, you become familiar with the workflow of editing annotations and XQuery functions.

Hans-Jürgen Rennau, hrennau@yahoo.de

Getting started with TopicTools (ttools)

vsn 2015-02-08-a

Welcome to a brief tutorial giving you an impression how to develop **topic tools**, which are command-line tools based on the `ttools` framework.

Prerequisites

You must have installed the BaseX XQuery processor. You can download it from here:

http://basex.org/products/download/all-downloads/

It is important to understand that BaseX is only required by the administrative operations of the `ttools` framework, in particular the initial creation of an application and the rebuilding steps which you launch after changing annotations. The topic tools, in whose development `ttools` assists you, are ordinary XQuery applications which can be run by any conforming XQuery processor.

Which processor to use for running the examples?

It is important to understand that the availability of certain features which `ttools` offers depends on the processor used to run your application. For example, when declaring input parameters of your application, certain parameter types are only available if the processor in use supports XQuery 3.0, and other types require support for the EXPath module `file`. When initializing an application, you explicitly or implicitly specify a **flavor** which determines the features of the XQuery processors expected to run the application.

When the intended processor is either Saxon or BaseX, you can specify the flavor by a simple string identifying the processor and its version.

Processor	**Version**	**flavor**
BaseX	7.9	basex79
	8.0	basex80
SaxonHE	9.5	saxonhe95
	9.6	saxonhe96
SaxonPE	9.5	saxonpe95
	9.6	saxonpe95
SaxonEE	9.5	saxonee95
	9.6	Saxonee96

When you use a different processor, the flavor should clarify two aspects:

- whether the processor supports XQuery 3.0
- whether the processor supports the EXPath `file` module

The flavor string then consists of the string xq10 (only XQuery 1.0) or xq30 (XQuery 3.0), optionally followed by character "f" indicating support for the `file` module. The flavor value should therefore be one of the following strings:

xq10, xq10f, xq30, xq30f

If you want to execute the examples of this tutorial, the recommended approach is to use the BaseX processor (version 7.9 or higher). This will ensure that the responses of ttools and your application look like described in the text.

Note also that several invocation examples use parameter types which are not available unless the processor supports the `EXPath` file module.

Scenario: topic tool `xwhite`

Suppose you want to create a command-line tool, named `xwhite`, which provides various operations for inspecting XML resources. A first version will support two operations, `names` and `paths`. The operations will report the names and data paths of elements and attributes encountered in a set of input documents.

Step #1 – creating the application

The first step of developing a topic tool is its initialization. You specify the tool directory, and (optionally) you also supply the name of a first module and the name(s) of first operations. Suppose the application directory is `/tta/xwhite`, the first module is `items.mod.xq` and the first operations will be `names` and `paths`. You initialize your application by the following call:

```
basex -b "request=new?dir=/tta/xwhite,mod=items,ops=names paths"
    /tt/bin/ttools.xq
```

`ttools:`

```
================================================================

Topic tool created:  xwhite
Tool directory:      /tta/xwhite

The tool can already be called. Example:
   basex -b "request=?" /tta/xwhite/xwhite.xq

Use operation 'add' for adding module prototypes. Example:
   basex -b "request=add?dir=/tta/xwhite, mod=fooMod, ops=fooOp barOp foobarOp"
             /tt/bin/ttools.xq
================================================================
```

Note

If you run the examples with a different XQuery processor than BaseX, you need to specify the flavor of the processor(s) expected to run the application. See above, "Which processor to use for running the examples" for details about the flavor. The flavor is supplied as the `flavor` parameter of the `new` operation. Example:

```
basex -b "request=new?dir=/tta/xwhitesaxon,mod=items,ops=names paths,
   flavor=saxonpe96" /tt/bin/ttools.xq
```

The flavor can later be changed by supplying a flavor parameter to the build function, for example:

```
basex -b "request=build?dir=/tta/xwhitesaxon,flavor=basex80"
         /tt/bin/ttools.xq
```

Step #2 – checking the `_help` operation

A topic tool is a collection of named operations, and each operation has a specific set of parameters. Formally, a topic tool has one single external variable named `request`, whose value is the invocation string which is parsed in order to extract the operation name and the parameter names and values.

The general invocation syntax of topic tools is:

```
      opName?foo=…, bar=…, …
```

where `opName` is the operation name and `foo` and `bar` are parameter names. If no parameters are specified, the invocation consists only in the operation name:

```
      opName
```

Every topic tool has a `_help` operation which reports the available operations and their parameters. The `_help` operation supports a shortcut syntax: an invocation string consisting of a single ? character is a shorthand for `_help`. A ? character followed by parameters is a shorthand for `_help?parameters`. First we call `_help` without parameters.

Call:
```
basex -b "request=?" /tta/xwhite/xwhite.xq
```

`xwhite:`

```
TOOL: xwhite

OPERATIONS     PARAMS
=======================================================================
_copyNcat      nodl?, query?, toNodl
_createNcat    nodl
_dcat          dfd+
_deleteNcat   nodl
_docs         dcat*, doc*, docs*, fdocs*
_doctypes     attNames, dcat*, doc*, docs*, elemNames, fdocs*, sortBy?
_feedNcat     dirs*, nodl
_help         default, mode, ops?, type
_nodlSample   model?
_search       nodl, query?
names         dcat*, doc*, docs*, fdocs*
paths         dcat*, doc*, docs*, fdocs*
======================================================================
```

All operations with a name starting with an underscore are inherited by the framework – they enhance your application before you write the first line of code. We want to try this out and request more detailed information about operations `_docs` and `_doctypes`. The parameters `type` and `default` request additional information about parameter types and default values, and the parameter `ops` restricts the report to operations with matching names.

Call:
`basex -b "request=?type,default,ops=_doc*" /tta/xwhite/xwhite.xq`

`xwhite`:

```
TOOL: xwhite

OPERATIONS   PARAMS
=================================================================================
_docs        dcat................ : docCAT* (sep=WS)
             doc................. : docURI* (sep=WS)
             docs................ : docDFD* (sep=SC)
             fdocs............... : docSEARCH* (sep=SC)
               At least 1 of these parameters must be set: dcat, doc, docs, fdocs

_doctypes    attNames=false...... : xs:boolean
             dcat................ : docCAT* (sep=WS)
             doc................. : docURI* (sep=WS)
             docs................ : docDFD* (sep=SC)
             elemNames=false..... : xs:boolean
             fdocs............... : docSEARCH* (sep=SC)
             sortBy=name......... : xs:string?; facets: values=name,namespace
               At least 1 of these parameters must be set: dcat, doc, docs, fdocs
=================================================================================
```

Step #3 – checking an inherited operation

It is somewhat amazing that our new topic tool already offers divers operations, although we have not yet written any code. We want to convince ourselves that those operations are indeed available. Therefore we call `_doctypes`, an operation reporting the “document types” (root element names and namespaces) of XML resources which can be specified in various way. We use the `docs` parameter of type `docDFD`, as this parameter type enables us to specify many documents in one fell sweep.

The type docDFD represents a directory filter, which is defined using a special filter syntax. (The letters DFD stand for “directory filter descriptor”). In summary, the filter string specifies one or more directories and one or more file name patterns. The filter prescribes on a per directory basis whether or not to include files from the direct and indirect sub directories. File name patterns are either positive or negative (including or excluding matching files). For further details (e.g. additional filters applied to sub directories) see the user manual. For example, the value

```
/a/b/c/*.xml *.xsd
```

selects all files with a name matching “*.xml” or “*.xsd” in directory /a/b/c . If a directory path is preceded by a | character, file search includes sub directories, so the value

```
|/a/b/c/*.xml *.xsd
```

searches for files in /a/b/c and its recursive sub directories. If you want to specify more than one directory, separate directories and file name patterns by a % character. For example, the value

```
|/a/b/c u/v % /*.xml *.xsd
```

searches the files not only in /a/b/c and its sub directories, but also in u/v (without its sub directories, as the path is not preceded by a | character). In order to exclude files matching some name pattern, insert a ~ character before the pattern. For example, the value

```
|/a/b/c u/v % /*.xml *.xsd  ~*test*
```

Ignores any files with a name including the string “test”. As these examples show, the parameter type docDFD can be very useful for specifying complex sets of input documents to be supplied to your applications.

For now, we request a report of all XML documents (pattern “*.xml”) found in the directory /projects/xq and its sub directories.

Call:
```
basex -b "request=_doctypes?docs=|/projects/xq/*.xml" /tta/xwhite/xwhite.xq
```

xwhite:

```
<z:doctypes xmlns:z="http://www.ttools.org/structure"
            countDocs="166" countDoctypes="24">
  <z:doctype name="company@" count="1">
    <doc href="file:///C:/projects/xq/workshop-2013-dev/examples/unit01-cave/doc-puzzle.xml"/>
  </z:doctype>
  <z:doctype name="msgType@http://www.ttools.org/logspy/ns/xquery-functions" count="59">
    <doc href="file:///C:/projects/xq/msgs-ttxml/AvailabilityAndPriceCheckRQ.xml"/>
    <doc href="file:///C:/projects/xq/msgs-ttxml/AvailabilityAndPriceCheckRS.xml"/>
    <doc href="file:///C:/projects/xq/msgs-ttxml/Booking_BookShoppingCart.xml"/>
        …

</z:doctypes>
```

This report lists the “document types”, represented by local name and namespace of the root elements, separated by a @ character, and for each observed document type lists all individual file names.

Step #4 – inspecting the function prototype

When creating the application, we already specified the names of a first application module (`items`) and the names of two first operations (`names` and `paths`). Therefore `ttools` generated an initial version of application module `items.mod.xq`, including two functions (`f:names` and `f:paths`) which will be called when the command-line call specifies operation `names` or `paths`, respectively. Now we want to test the prototype of operation `names`. We consult again the `_help` operation in order to learn the names and types of the operation parameters.

Call:
```
basex -b request=?type,default,ops=names /tta/xwhite/xwhite.xq
```

`xwhite`:
```
TOOL: xwhite

OPERATIONS   PARAMS
===============================================================================
names        dcat................ : docCAT*  (sep=WS)
             doc................. : docURI*  (sep=WS)
             docs................ : docDFD*  (sep=SC)
             fdocs............... : docSEARCH*  (sep=SC)
               At least 1 of these parameters must be set: dcat, doc, docs, fdocs
===============================================================================
```

The response shows that operation `names` requires at least one of the parameters `doc`, `docs`, `dcat`, `fdocs`. This is, again, somewhat surprising – how can `ttools` anticipate the names and types of operation parameters, not to mention subtle constraints like “at least one of these parameters …”? When generating the prototype of an application module, `ttools` generates for each requested operation (`names` and `paths`, in our case) initial **annotations** declaring a set of parameters which is often useful. We are expected to change and extend these annotations, but the generated annotations provide us with a convenient starting point. And the initial annotations enable us to immediately “test” the generated prototype of our operation.

As we already know the data type `docDFD`, we use in our first test the parameter `docs`.

Call:
```
basex -b "request=paths?docs=|/projects/xq/*.xml" /tta/xwhite/xwhite.xq
```

`xwhite`:
```
<z:names xmlns:z="http://www.ttools.org/xwhite/ns/structure" countDocs="166"/>
```

The response is a `z:names` element reporting the number of supplied documents. We inspect the generated code which produces the output:

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat fdocs')
    return
        <z:names countDocs="{count($docs)}">{
           ()
        }</z:names>
};
```

The application code does not receive file names – it receives the document nodes themselves, corresponding to these file names. The supplied *parameter text* is a string describing the filtering of directories. The delivered *parameter value* is a sequence of document nodes. (The generated code does not prove this, but it will become obvious when we start to edit the code, using `$docs` as the first step of path expressions.)

This little test can be used to illustrate the basic processing model of topic tools. When a tool operation is called, generated code does two things:

- It *parses* the request text and fills a request element representing the user input
- It *calls* the function implementing the requested operation, passing the request element to the function

The called function can retrieve the operation parameters (`foo` and `bar`, in the example) by calling a `ttools`-provided retrieval function, `tt:getParam` or `tt:getParams`

The binding of operation names to functions is established by annotations. Strictly speaking, these are pseudo annotations – XQuery comments observing a particular syntax. The generated module contains the following annotations:

```
(:~@operations
   <operations>
      <operation name="names" type="node()" func="names">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep="SC" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <param name="fdocs" type="docSEARCH*" sep="SC" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
      <operation name="paths" type="node()" func="paths">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep="SC" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <param name="fdocs" type="docSEARCH*" sep="SC" pgroup="input"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
    </operations>
```

Annotations of type @operations declare operations which the containing module contributes to the topic tool. Each operation is represented by an <operation> element, with @name and @func attributes specifying the operation name and the name of the implementing function. Each operation parameter is defined by a <param> child of the <operation> element. Parameter name and type are conveyed by the @name and @type attributes on <param>. When a parameter type allows multiple value items, the @sep attribute specifies the item separator – either WS for whitespace, or SC for semicolon. An optional @pgroup attribute declares membership in a parameter group, which is represented by a <pgroup> element. This element enables the module author to specify cardinality constraints which do not apply to a single parameter, but to a set of parameters. For example, the generated annotations specify that at least one parameter of the group `input` must be supplied.

You implement a tool operation by providing

(a) an annotation element (<operation>) specifying the name of the operation and the names and types of all operation parameters
(b) a function implementing the operation, which has a single parameter

Within the function, you retrieve the operation parameters by passing the request element to a `ttools`-provided function - `tt:getParam` for the retrieval of a single parameter (which may be multi-valued) and `tt:getParams` for the aggregated retrieval of several parameters. So the expression

```
        tt:getParams($request, 'doc docs dcat fdocs')
```

returns all documents specified by any of the parameters `doc`, `docs`, `dcat` and `fdocs`. To check this aggregation, we repeat our call with an additional parameter `doc`, typed `docURI*`. The parameter value must consist of one or several document URIs, separated by whitespace.

Call:
```
basex -b "request=paths?docs=|/projects/xq/*.xml,
                        doc=/projects/infospace/pcol/countsTree.xml"
   /tta/xwhite/xwhite.xq
```

`xwhite`:

```
<z:paths xmlns:z="http://www.ttools.org/xwhite/ns/structure" countDocs="167"/>
```

The result is as we expected.

Step #5 – writing a first version of an operation

So far we have not written a single line of code. Now we create a first non-dummy version of operation `names`. The goal is to list all element names and all attribute names encountered in the supplied set of input documents. We replace the generated function code by the following code:

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat fdocs')

    let $enames := distinct-values($docs//*/local-name(.))
    let $anames := distinct-values($docs//@*/local-name(.))
    return
        <z:names countDocs="{count($docs)}">{
            <z:enames count="{count($enames)}">{
                for $n in $enames order by lower-case($n)
                return <z:ename>{$n}</z:ename>
            }</z:enames>,
            <z:anames count="{count($anames)}">{
                for $n in $anames order by lower-case($n)
                return <z:aname>{$n}</z:aname>
            }</z:anames>
        }</z:names>
};
```

In order to test the new version, we repeat our call of operation `names`.

Call:
```
basex -b "request=names?docs=|/projects/xq/*.xml" /tta/xwhite/xwhite.xq
```

`xwhite`:

```
<z:names xmlns:z="http://www.ttools.org/xwhite/ns/structure" countDocs="166">
  <z:enames count="692">
    <z:ename>Absolute</z:ename>
    <z:ename>AccomCode</z:ename>
    <z:ename>AccomIds</z:ename>
    …
    <z:ename>XMLMandant</z:ename>
    <z:ename>XMLMandantPassword</z:ename>
    <z:ename>YearOverview</z:ename>
  </z:enames>
  <z:anames count="252">
    <z:aname>AbsoluteSurcharge</z:aname>
    <z:aname>Action</z:aname>
    <z:aname>Age</z:aname>
    …
    <z:aname>Weightage</z:aname>
    <z:aname>XCoord</z:aname>
    <z:aname>YCoord</z:aname>
  </z:anames>
</z:names>
```

By now we have reached the point where a first operation is available, delivering some useful result. Now we shall refine the function by adding further parameters.

Step #6 –adding a parameter

We want to give the tool user the possibility to choose whether elements and attributes, or only elements, or only attributes are reported. We first extend the generated annotation by a further parameter `scope`, which we declaratively constrain to have one of the values `elem`, `att`, `all`, defaulting to `all`. This is accomplished by adding a further <param> element to the <operation> element of operation `names`:

```
<operation name="names" type="node()" func="names">
   <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
   <param name="docs" type="docDFD*" sep="SC" pgroup="input"/>
   <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
   <param name="fdocs" type="docSEARCH*" sep="SC" pgroup="input"/>
   <param name="scope" type="xs:string?" fct_values="all,elem,att"
                                          default="all"/>
   <pgroup name="input" minOccurs="1"/>
</operation>
```

Note the @fct_values attribute defining a list of valid parameter values. Also note that the parameter is optional (type is `xs:string?`), but has a default value (see @default attribute). It is important to know that the application can *rely* on the framework to enforce any declared facets – it will not even call the application code in case of violations, and it supplies the user with detailed error information.

Now we adapt the function code.

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat fdocs')
    let $scope as xs:string := tt:getParams($request, 'scope')

    let $enames := distinct-values($docs//*/local-name(.))
    let $anames := distinct-values($docs//@*/local-name(.))
    return
        <z:names countDocs="{count($docs)}" scope="{$scope}">{
            if ($scope eq 'att') then () else
            <z:enames count="{count($enames)}">{
                for $n in $enames order by lower-case($n)
                return <z:ename>{$n}</z:ename>
            }</z:enames>,
            if ($scope eq 'elem') then () else
            <z:anames count="{count($anames)}">{
                for $n in $anames order by lower-case($n) return
                <z:aname>{$n}</z:aname>
            }</z:anames>
        }</z:names>
};
```

Note that the application code can also rely on the default value declaration – otherwise, the type declaration `xs:string` would cause an error when no parameter was specified.

Before we can test the code, we must rebuild the tool. This is necessary because we changed the annotations. The rebuild is achieved by calling `ttools` with operation `build` and passing the tool directory as parameter `dir`:

```
basex -b request=build?dir=/tta/xwhite /tt/bin/ttools.xq
```

Now we test our new version of names, calling it with parameter `scope` set to the value `att`.

Call:

```
basex -b "request=names?docs=|/projects/xq/*.xml,scope=att"
   /tta/xwhite/xwhite.xq > anames.xml
```

`xwhite`:

```
<z:names xmlns:z="http://www.ttools.org/xwhite/ns/structure"
         countDocs="166" scope="att">
  <z:anames count="252">
    <z:aname>a</z:aname>
    <z:aname>AbsoluteSurcharge</z:aname>
    <z:aname>Action</z:aname>
    <z:aname>active</z:aname>
    <z:aname>afterCombinatorics</z:aname>
    ..
  </z:anames>
</z:names>
```

If we omit the parameter, again all names are delivered, due to the default value `all`.

Step #7 –adding a smart parameter

The `scope` parameter which we just added has a built-in type, `xs:string`. The `ttools` framework defines a number of "smart" parameter types characterized by an *enhancement* which the framework applies to the original parameter value. The type `docDFD` which we already discussed is a good example of this principle: the parameter text consists of a filter string, but the delivered parameter value is a set of document nodes which framework code obtained by parsing the filter, searching the file system accordingly and transforming the matching file names into document nodes.

Another smart parameter type is `nameFilter`. The syntax is a filter applicable to names, and the delivered parameter value is a structured representation of that filter (a <nameFilter> element). This element is "valuable", as it can be passed to framework-provided functions – e.g. `tt:matchesNameFilter` – performing a sophisticated filtering operation. We now add an optional operation parameter which enables the user to restrict the report to names matching a name pattern. So we add to the annocations a parameter `names`, like so:

```
      <operation name="names" type="node()" func="names">
         <param name="doc" type="docURI*" sep="WS" pgroup="input"/>
         <param name="docs" type="docDFD*" sep="SC" pgroup="input"/>
         <param name="dcat" type="docCAT*" sep="WS" pgroup="input"/>
         <param name="fdocs" type="docSEARCH*" sep="SC" pgroup="input"/>
         <param name="scope" type="xs:string?" fct_values="all,elem,att"
                                               default="all"/>
         <param name="names" type="nameFilter?"/>
         <pgroup name="input" minOccurs="1"/>
      </operation>
```

And we refine the function code as follows:

```
declare function f:names($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat fdocs')
    let $scope as xs:string := tt:getParams($request, 'scope')
    let $names as element(nameFilter)? := tt:getParams($request, 'names')

    let $enames :=
      distinct-values($docs//*
               [tt:matchesNameFilter(local-name(.), $names)]/local-name(.))
    let $anames := distinct-values($docs//@*
               [tt:matchesNameFilter(local-name(.), $names)]/local-name(.))
    return
        <z:names countDocs="{count($docs)}" scope="{$scope}"
                 nameFilter="{$names/@text}">{
            if ($scope eq 'att') then () else
            <z:enames count="{count($enames)}">{
                for $n in $enames order by lower-case($n)
                return <z:ename>{$n}</z:ename>
            }</z:enames>,
            if ($scope eq 'elem') then () else
            <z:anames count="{count($anames)}">{
                for $n in $anames order by lower-case($n)
                return <z:aname>{$n}</z:aname>
            }</z:anames>
        }</z:names>
};
```

As we changed the annotations, we first rebuild the tool:

```
basex -b request=build?dir=/tta/xwhite /tt/bin/ttools.xq
```

and then test it by supplying a name filter selecting names containing either “charge” or fare”, but excluding names ending with “details”.

Call:
```
basex -b "request=names?docs=|/projects/xq/*.xml,
   names=*fare* *charge* ~*details"
   /tta/xwhite/xwhite.xq > names.xml
```

`xwhite:`

```
<z:names xmlns:z="http://www.ttools.org/xwhite/ns/structure"
         countDocs="166" scope="all"
         nameFilter="*fare* *charge* ~*details">
  <z:enames count="5">
    <z:ename>FareBase</z:ename>
    <z:ename>NettoFare</z:ename>
    <z:ename>PreferredFare</z:ename>
    <z:ename>ServiceCharge</z:ename>
    <z:ename>Surcharge</z:ename>
  </z:enames>
  <z:anames count="4">
    <z:aname>AbsoluteSurcharge</z:aname>
    <z:aname>FareType</z:aname>
    <z:aname>RelativeSurcharge</z:aname>
    <z:aname>SurchargeOrder</z:aname>
  </z:anames>
</z:names>
```

Step #8 – tackling a second operation

Now we quickly provide a first implementation of operation `paths`. We replace the generated code of function `f:paths` by a first version returning a list of all data paths:

```
declare function f:paths($request as element())
        as element() {
    let $docs := tt:getParams($request, 'doc docs dcat fdocs')
    let $paths :=
        for $item in $docs//(*,@*)
        group by $path := f:_path($item)
        order by lower-case($path)
        return <z:path count="{count($item)}">{$path}</z:path>
    return
        <z:paths countDocs="{count($docs)}" countPaths="{count($paths)}">{
           $paths
        }</z:paths>
};

declare function f:_path($n as node())
        as xs:string {
  string-join(
    $n/ancestor-or-self::node()/concat(self::attribute()/'@', local-name())
    [string()], '/')
};
```

Note that we added also an auxiliary function (`f:_path`) which does not implement an operation.

As we did not (yet) change any annotations, we need not rebuild the tool and immediately proceed to test it.

Call:

```
basex -b "request=paths?docs=|/projects/xq/*.xml" /tta/xwhite/xwhite.xq
```

`xwhite:`

```
<z:paths xmlns:z="http://www.ttools.org/xwhite/ns/structure"
         countDocs="166" countPaths="6841">
  <z:path count="4">commands</z:path>
  <z:path count="4">commands/create-db</z:path>
  <z:path count="4">commands/create-db/@name</z:path>
  <z:path count="4">commands/drop-db</z:path>
  <z:path count="4">commands/drop-db/@name</z:path>
  <z:path count="1">company</z:path>
  …
</z:paths>
```

By now you have a good grasp of the basic workflow involved in the iterative refinement of operations:

(a) adding a parameter by editing the annotations
(b) editing the code, using the new parameter
(c) rebuilding the tool via `build` call and
(d) testing the new version, supplying values to the new parameter

You would have no difficulties in refining the paths operations, should you wish to – for example offering support for name filters.

Step #9 –adding a new operation

Suppose you want to add a new operation for which you do not yet have a prototype. There are two possibilities: the new operation shall be located in a new application module, or it shall be added to an existing module. In the first case, the easiest approach is to let `ttools` generate a new application module, using its `add` operation:

Call:

```
basex -b "request=add?dir=/tta/xwhite,mod=trees,ops=stree vtree ctree"
   /tt/bin/ttools.xq
```

`ttools:`

```
=========================================================
XQuery module created: trees.mod.xq
Operations:            stree vtree ctree

Directory:             /tta/xwhite
Topic tool:            xwhite
```

```
The new operations are available. Example:

   basex -b "request=stree?doc=doc1.xml doc2.xml doc3.xml" /tta/xwhite/xwhite.xq

To implement them, edit these functions: stree vtree ctree
===============================================================
```

Then you proceed exactly as you proceeded when working with the initial module.

If the new operation should be located in an existing module, `ttools` cannot be used in order to generate prototypes of annotations and code. The recommended way is copy and paste:

(a) some <operation> element from the annotations, adapting the operation name and the function name (@name and @func)
(b) some function which implements an operation

From then on you proceed in the same way as when working with a generated prototype.