BaseX for dummies

A short introduction to BaseX in general, and to the BaseX / PHP combination in particular by *Paul Swennenhuis – paul@swennenhuis.nl*

Introduction

About a year ago I was looking for tools to produce pivot tables out of loosely structured XML data. And yes, I found BaseX. It was a JAVA solution - I did not care much about JAVA solutions at that time, and if I were to install it via Unix commands and a Web configuration file I'm sure I would have abolished the procedure, but hey, BaseX uses a simple Windows installer, and loading the GUI was as simple as getting two children to have a fight over the last piece of chocolate-pie - but it used a GUI and had a pre-configured database available for testing so I went ahead anyway.
I was immediately struck by the Visualization options, especially the Map and Plot ones. Great ways to visualize data!
After some attempts I figured out how the Plot visualization works. At first the map of the world plot that can be seen here http://docs.basex.org/wiki/File:Scatterplot.jpg felt like magic to me, until I realized what was being plotted. I realized that I could very quickly make all the pivots that I'd ever want with BaseX. And with a very good performance as well.

So I found BaseX but then I forgot about it almost immediately as there were other matters that needed my attention.
A few weeks ago however I was asked to think about a solution to store and query semi-structured XML data. I remembered that I had downloaded a product that would make a good candidate, but what was the name again...? Oh yes, BaseX.
So again I downloaded BaseX. But now I had a closer look at it. I read the manuals (well, most of them), I read the excellent thesis of the founder of BaseX himself, Christian Grün. [Really, you should give it a try. It is excellent reading material. You can find it here: http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-opus-127142/Dissertation_Gruen_2010.pdf?sequence=1]
Anyway, I was quite excited about BaseX and I was convinced it would be the perfect solution for the problem I needed to solve.
But wait... wasn't BaseX a JAVA product? So didn't I need a (web)server with JAVA enabled? While all I had ever worked with were these bread-and-butter PHP servers... So how to proceed?

I decided to first try to get comfortable with BaseX. How do queries work? How do you call BaseX from outside the GUI? How to work with the server/client setup? How can you setup and manage XML databases from within PHP scripts?
So here's my layman's report of trying to setup BaseX, PHP and JavaScript in a way that I feel comfortable with and that I can manage. I hope it can be of use for other developers that struggle with this subject.

In the follow-up of this introduction I will dive into the part of setting up BaseX on a server that supports JAVA.

Not everything in this introduction will be the most elegant or the most efficient solution for the problems presented. Sorry for that. I have this inclination to find solutions that work, and only then am I prepared to look beyond...

How to use this introduction

This introduction is meant to be a practical guide for setting up a website or application that uses BaseX as underlying database. It more or less follows my own learning steps in getting to know BaseX and XQuery. These steps, presented in the following *Step by step* section, hopefully will help you to get comfortable with BaseX.
In the *Appendix* I discuss a few related topics that did not fit very well in the *Step by step* section.

Step by step

Use the GUI to create a XML database

The GUI of BaseX is a very good starting point for getting to know BaseX. It is also the perfect place to create your XML database. I am assuming here that you know what XML data you want to store and query. If not, you can skip this step and work with the Facts database that is provided in the distribution package of BaseX.

To create a database:

1. Make sure you have a valid XML file or a folder with a bunch of valid XML files (preferably but not necessarily sharing the same structure)
2. In the BaseX GUI, select Database > New to create a new database
3. Browse to the XML file or to the folder with XML files that you want to put in the database, and give the database a name.
4. Click OK to create the database

> *Some important notes:*
> 1. *Creating a database in BaseX does NOT mean that from now on all changes in the selected XML file(s) are automatically reflected in the database. It just means that you made the XML file(s) queryable for BaseX.*
>   *You cannot even update the database with the XML file(s) manually using Database > Properties > Add. It will ADD the contents of the file(s) to the database, not update them! Updating BaseX databases is done with database commands OR XQueries. More information on how to update databases can be found in the Appendix.*
> 2. *You can also add files in JSON format. However, this has some drawbacks that will also be explained later.*

2. XPath exercises

Now that you created a database you can execute some XPath expressions on it to get yourself acquainted (again?) with how they work. Start off with simple expressions and gradually increase the complexity.
Set the Search Bar mode to XQuery and then enter your queries / XPath expressions.

Notice how the Map visualization lights up the search results when there are hits.

As I do not know what data are in your database I cannot give example queries to test. Therefore I list some queries that you can execute on the Facts database

a. All cities in the database

```
//city
```

b. All cities that have a @country attribute

```
//city[@country]
```

c. All cities that have a @longitude greater than 20

```
//city[@longitude > 20]
```

d. The value of the population tag of all cities that have a @longitude greater than 20

```
data(//city[@longitude > 20]/population)
```

Note: without the data() function the complete tag will be displayed, not just the text

e. The name of the cities that have a population of over 100000 inhabitants

```
//city[number(population)>100000]/name
```

f. The countries whose name starts with a P (case insensitive) and have a population of over 1.000.000

```
//country[matches(@name,'^P','i') and @population>10000000]
```

> *Did you know about the matches() function in XQuery?*
> *When I learned XML and XSLT back in 2001 or so text matches were a real pain in the ass.*
> *You could test for equality, for "starts-with" or "ends-with" and you could get a substring and that was about it. Imagine having to check if a string is a valid email-address...*
> *But XQuery supports regular expression with the matches() function.*
> *I used it to do a real nice full text lookup in a FLWOR expression that will be discussed in the last step of this section.*

3. Start blooming

Now let's take it a step further and use FLWOR expressions[1]. This is an important step in getting prepared for retrieval of XML data with PHP scripts (or AJAX calls for that matter).
We leave the Search Bar of the GUI and turn to the Editor. (If it is not visible press Ctrl-E to show it).
Copy the last successful query of step 2 in the editor, and precede it with this line

```
let $results :=
```

and then add this line:

```
return <results>{$results}</results>
```

Then enter the Go button (green arrow).
Voila, you wrote your first XQuery!

[1] Refer to the *Appendix* section for a few notes on FLWOR and XQuery

(If you are unsuccessful, try this one (with the Facts database opened!):

```
let $results := //country[matches(@name,'^P','i') and
@population>10000000]
return <results>{$results}</results>
```

Now close the currently opened database and try to run the XQuery again.
It will fail. Obviously, //country can only be evaluated against a XML set of data, be it a fragment, a XML document, multiple XML documents or a database.
So let's change the query in such a way that it will always succeed - provided that the Facts database exists.

```
let $results := collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
return <results>{$results}</results>
```

> *You could also have used db:open("Facts") instead of collection("Facts") but I prefer the collection() function because db:open() is BaseX-specific.*

In this XQuery we used merely the L (Let) and R (Return) parts of FLWOR.
Let's add in the F (For) by modifying the query above:

```
for $country in collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
let $province := count($country/province)
return <country><name>{$country/@name}</name>
<provinces>{$province}</provinces>
</country>
```

This will yield a list of the names of countries starting with 'P' along with the number of provinces of that country.

Suppose we want the list to be ordered on country name; here comes in the O of Order by:

```
for $country in collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
let $province := count($country/province)
order by $country/@name
return <country><name>{$country/@name}</name>
<provinces>{$province}</provinces>
</country>
```

Finally we add a condition using the W (Where) of FLWOR

```
for $country in collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
where number($country/@infant_mortality) le 50
let $province := count($country/province)
order by $country/@name
return <country><name>{$country/@name}</name>
<provinces>{$province}</provinces>
</country>
```

Now hold on to that last XQuery. What it should yield is this:

```
<country>
  <name name="Philippines"/>
  <provinces>0</provinces>
</country>
<country>
  <name name="Poland"/>
  <provinces>49</provinces>
</country>
```

A perfectly fine XML result.
But... what if you were to return this result to a XML parser (for example PHP's SimpleXML)?
Exactly, it will complain about a missing root element.
So, somehow we have to enclose the result in a root element.
Something like this:

```
<countries>
  <country>
    <name name="Philippines"/>
    <provinces>0</provinces>
  </country>
  <country>
    <name name="Poland"/>
    <provinces>49</provinces>
  </country>
```

```
</countries>
```

However, we can not add the <countries> tag in the R (Result) part:

```
for $country in collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
where number($country/@infant_mortality) le 50
let $province := count($country/province)
order by $country/@name
return <countries><country><name>{$country/@name}</name>
<provinces>{$province}</provinces>
</country></countries>
```

The <countries> tag will be added for each and every country found, just like the <country> tag, so there still is no single root element.
How about surrounding the FLWOR with an additional <countries> tag then?
Well, in fact that IS the solution. However, it took me a considerable amount of time to find out that you have to enclose the FLWOR expression itself with curly braces in order to make it work:

```
<countries>
{
for $country in collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
where number($country/@infant_mortality) le 50
let $province := count($country/province)

order by $country/@name
return <country><name>{$country/@name}</name>
<provinces>{$province}</provinces>
</country>
}
</countries>
```

Nice!

4. Approach BaseX from the outside

It's time for our next step: use BaseX from the outside, i.e. not as a standalone application but as a client/server architecture.

So close the GUI app, then open a command window (Windows Start menu, enter "cmd" + enter)
Browse to the folder where you installed BaseX, usually C:\Program Files (x86)\Basex, **cd** to the bin subfolder and enter basexhttp.bat
This will start both the BaseX database server (on port 1984) and the BaseX HTTP server (on port 8984)
In fact, using another approach, it would have been sufficient to just start the database server (basexserver.bat), and use a BaseX client - for example this one: https://github.com/BaseXdb/basex/tree/master/basex-api/src/main/php - to execute queries on the database, but in my opinion this results in a clumsy and difficult to maintain system. (Elaborations on this approach can be found in step *5. PHP*)

OK, now let's see if the HTTP server is running: enter `http://localhost:8984/` in your favourite browser (Chrome or Firefox). You should be asked for a username and password. Enter the credentials of one of the accounts listed below:

a) the special, limited BaseX database user account you setup before
b) the BaseX database admin account with password modified by you
c) the original, unmodified BaseX admin/admin account

> *The crowd goes "Hurray!" if you used option a)*
> *It goes "Well, ok then" if you used option b)*
> *And it yells "Booohh!" if you used option c)*
>
> *See "Creating and modifying BaseX database accounts" in the Appendix to learn how to add and/or modify accounts. And yes, you really should avoid using the admin account.*

Now the default homepage should show, starting off with "BaseX HTTP Services Welcome to the BaseX HTTP Services, which allow you to...".
If not, move all the files in the BaseX folder that you modified after installing BaseX, especially any *.xq files that you have created, to another location and try again.

To be frank - and, I admit, a bit harsh to the person that created it - this welcome page is too intimidating for developers who just managed to display it in the first place.

```
%rest:path("/hello/{$world}")
```

?? Sorry, what is that supposed to mean or do?

Anyway, if the page shows you know the HTTP server is working.
One thing we can do now - it seems like magic - is list all the resources that were used to compose a particular BaseX database, e.g. the Facts database:

```
http://localhost:8984/rest/Facts
```

It uses the REST interface of BaseX. Probably you have heard of REST and, unlike me, know exactly what that stands for and how to set-up a REST API in PHP.
If not, don't worry. Just know that we'll be using it from now on.

OK, now let's see if we can run the XQueries we used in the previous step and have the HTTP server present the results.

ONE IMPORTANT NOTE OF WARNING - and I can not stress this enough - : always ensure that the query result has a SINGLE root element!

Now create a text file in your favorite editor - it should be UltraEdit - and paste the content of the query I listed earlier in it:

```
<countries>
{
for $country in collection("Facts")//country[matches(@name,'^P','i')
and @population>10000000]
where number($country/@infant_mortality) le 50
let $province := count($country/province)

order by $country/@name
return <country><name>{$country/@name}</name>
<provinces>{$province}</provinces>
</country>
}
</countries>
```

and save the file in `[path to BaseX]/webapp`.
Call it countries.xq

Then enter this address in the browser:
`http://localhost:8984/rest/?run=countries.xq`

Wow! The HTTP server presents exactly the same result as the BaseX GUI. How sweet life can be!

But what - I hear you thinking - what if you do not want countries starting with 'P' but with 'N'?
Yes, of course you can edit the countries.xq file and change ^P to ^N
But clearly you want to be able to change it to ANYTHING. In other words, you want to use a parameter.
No problem. In countries.xq, change the "'^P'" part to $country, and add this line before it:

```
declare variable $country as xs:string external;
```

Then change the browser URL to
`http://localhost:8984/rest/?run=countries.xq&country=^N`
or
`http://localhost:8984/rest/?run=countries.xq&country=s$` (i.e. countries ending with 's')

> *Note: if you now run countries.xq in the GUI of BaseX you will get an error saying $country is not defined. To prevent this you can provide a default value:*
> *declare variable $country as xs:string external := '^P';*
> *This way the query will be valid both in the GUI and when run in the browser.*

That was easy. So now we know how to make BaseX run predefined queries, and how to make them dynamic using parameters. (more about XQuery variable types can be found at http://docs.oracle.com/cd/E13214_01/wli/docs92/xref/xqdtypes.html)

Let's turn to the PHP side of things.

5. PHP

As mentioned in the previous part, we could also have chosen to use a BaseX PHP client that queries the database directly, i.e. without using the REST interface.
I chose not to, for several reasons. The most important one of them I will mention later. Here I would like to point out that in order to use the BaseX PHP client you would have to compose the XQueries in PHP. In light of the "separate code from content" pattern, I don't think it is wise to compose and store queries in PHP and then send them to the BaseX server. You could of course create a database of XQueries, or save them in a text file, but then you'd still loose on another point: if you store the queries in a place that BaseX can find, you can also use them from within JavaScript, or .NET, or whatever programming environment you decide to throw against it.
The URL that we used above would then also be usable in other environments.
So I chose to work with .xq files, and store them in the webapp folder of BaseX.

OK, let's make a PHP scripts that calls the URL above and prints the results.
That seems easy enough:

```
  <?php
   ...
   ...
   $result =
file_get_contents("http://localhost:8984/rest/?run=countries.xq&coun
try=s$"));
   var_dump ($result);
   ...
   ...

  ?>
```

However, what we forgot here is that the REST interface uses basic authorization (that's why we had to sign in when we asked the browser to display `http://localhost:8984/`) and without it the request will return an “unauthorized” error.
So, we have to figure out how to send that authorization information together with the request.
At first I thought that curl would be the way to go (and I am sure it can be done with curl) but some forum pointed me to another solution that I found quite elegant. It appears that file_get_contents() has an optional parameter that can be used to set HTTP headers. I used this to set the basic authentication of BaseX (the variable named $opts in the code example below), then created a $context from that and finally used that as $context parameter for file_get_contents().
See the code below where, in addition, the URL has been made more generic and copies the request URI parameters in the BaseX request.

```
   // credentials
   $username = "admin";
   $password = "a33445";

   // Create a stream
   $opts = array(
        'http'=>array(
          'method' => "GET",
          'header' => "Authorization: Basic " .
base64_encode("$username:$password")
        )
      );

      $context = stream_context_create($opts);

   // Open the file using the HTTP headers set above

    $result = file_get_contents("http://localhost:8984/rest?%s",
$_SERVER['QUERY_STRING']), FALSE, $context);
```

> *The $_SERVER['QUERY_STRING'] adds in the parameters that are present in the URL to call the PHP script.*

... and now you may wonder: why use PHP to do a BaseX request? Why not call that URL directly? After all you're using the same parameters...

True. The main reason for using a PHP script is that I have to convert the XML results of BaseX to JSON and return them as a AJAX JSONP result.

I DO know that BaseX has its own JSON converters (called serializers) but to be honest, I do not like them.
The "JSONML" way of BaseX JSON serialization outputs both tags and attributes in the same way so you can not discern the two of them anymore. Also, it outputs the name of an element as an array entry, not as an object key.
For example, <a>c will result in ["a",["b","c"]]
where I would have expected {"a":{"b":"c"}}
To get this last form I would have to use the "direct" way of BaseX JSON serialization, but in that case I would have to inform the serializer about the types of ALL possible nodes:

json:serialize(<json type="object" objects="a"><a>c</json>)
which will indeed produce {"a":{"b":"c"}}
But hey, I do not feel like I should tell a serializer how to treat data. It should figure that out for itself.

Anyway, after experimenting a bit with BaseX serializers I decided to keep it plain and simple and have BaseX output just plain XML, and do the JSON serialization myself in PHP.
This also gave me the opportunity to create a JSONP return value with the serialized data so it could be used in an AJAX call (remember that cross-origin AJAX calls are NOT allowed, so if the BaseX server is not at the same location as your PHP server AND uses the same ports you will have to use JSONP, or allow cross-domain requests using CORS):

```
// convert the XML results from BaseX to JSON
...
// return a JSONP result, i.e. with a callback
print ($_GET['callback'] . '('.json_encode($result).')');
```

The full, generic “perform a BaseX query, convert the results to JSON and return a JSONP result” PHP script can be found in the basexfordummies folder in the package.

6. Putting it all together

Using the techniques described in the previous sections we can now create a real-life example.
You can find all the files needed for this example in the basexfordummies folder in the package.
We're going to write a "display while you type" search page, where results are presented immediately while you are typing, similar to what Google does for the suggestions that are presented when you enter search terms. Go to Google, press F12 to display the debugging tools, switch to the Net tab, and start typing in the search box to see what I am referring to.

We will be searching in the Facts database. Of course you can apply the presented solution to other XML databases.

The XQ query file

We want to do a full text search on the Facts database, and we want the search to be configurable, obviously.
The query we will be using, explained further on, is:

```
declare variable $search as xs:string external := "city" ;
<found>
{
let $found := collection("Facts")/descendant-or-
self::*[matches(text(), $search, "i")]
for $x in $found
order by $x/text()
return
<parent type='{data($x/ancestor-or-self::*[matches(local-name(),
"country|city|province|religions|population") ][1]/node-name())}'>
  <result> {$x}</result>
</parent>

}
</found>
```

The first line declares our $search variable with a default value of “city”.
The Let part of the query was formed by looking how BaseX optimized a query for a simple search from the Search bar (in the Query info box).
It uses the descendant-or-self axis for the nodes to search, and the matches() function to do a (simple) case-insensitive regular expression search.

```
let $found := collection("Facts")/descendant-or-
self::*[matches(text(), $search, "i")]
```

Search results are sorted in order of the found text:

```
order by $x/text()
```

The Return part is quite interesting. What I wanted to do is not just show the found text, but also the context node of that text.
I could have opted for a plain parent (..) context but that would almost always have shown "name". So I decided to trace the ancestors node upward to find the context node whose name is in a list of "acceptable" tags: country, city, province, ...
So every match returns both the found text and the context node.
Making the final query as shown above.

Notice that the [1] modifier in the Result part ensures that only the nearest ancestor of the node with the found text in the "acceptable" tags list is returned.
Also note that this is a relatively expensive operation: all ancestors of the found node have to be explored and only the first one gets selected.

Nonetheless the query executes quite fast, thanks to the highly optimized indexes of BaseX. I seldom see execution times higher than 50 msecs, provided that the length of the search-text is at least 3 characters.
(For a discussion on how this is achieved I heartily recommend reading Christian Grüns thesis *Storing and Querying Large XML Instances* at http://kops.ub.uni-konstanz.de/bitstream/handle/urn:nbn:de:bsz:352-opus-127142/Dissertation_Gruen_2010.pdf?sequence=1)

Also notice that the calling script has to provide a "search" parameter in the URL.

Save the XQuery as fulltext.xq in the webapp folder of BaseX (or, if you really want to be organized, in a sub-folder thereof).

HTML Page

We already created a generic PHP script to run a query against BaseX – and we will be using that script - so all we need for our real-life example is a HTML page with JavaScript code that enables the user to enter a search term, catches the users search input, uses AJAX to call the generic PHP script with the correct parameters and presents the results.

Open the HTML page `search.html` in the package in the editor of your choice and take a look at it. It uses JQuery - as a developer I assume you are familiar with JQuery - but could be rewritten in plain JavaScript.
The comments in the file should explain to you what is going on. If not, drop me an e-mail.

Now load `http://localhost/basexfordummies/search.html` in your browser and start typing. When the number of characters entered exceeds 2 you should get results.

Summary

1. Start the BaseX servers
   Windows cmd >> go to the BaseX "bin" directory >> basexhttp.bat
2. Create the fulltext.xq file and save it in the webapp folder of BaseX or a subfolder thereof
3. Create the BaseXRest.php file and place it in the `basexfordummies` folder of localhost
4. Create the `search.html` file and also store it in the `basexfordummies` folder of localhost
5. Make sure that the XML to JSON converter `XMLToJSON.class.php` is also present in the `basexfordummies` folder
6. Enter `http://localhost/basexfordummies/search.html` in the browser
7. Enter search terms and examine the presented results

Appendix

Updating BaseX databases

Basically there are two ways of updating BaseX databases: directly with database commands, or indirectly using XQuery. The last is more flexible because you can tell XBase WHERE to insert the XML.

For both solutions you will need a (PHP) database client, for example the one located at https://github.com/BaseXdb/basex/tree/master/basex-api/src/main/php

First, start the BaseX database server.
(Windows cmd >> go to the BaseX "bin" directory >> basexserver.bat)
Next, start your PHP localhost HTTP server of choice (I use EasyPHP)
Next, create a PHP file that starts a BaseX session, opens the database of choice, and either issues a ADD command or inserts a XML file or fragment to it: ($result contains a XML fragment (as a string))

```
function addToDatabase($result) {
    // create session
    $session = new Session("localhost", 1984, "paolo", "a33445");
    // open database
    $session->execute("open quaynResults");

    $session->execute(sprintf("xquery insert node %s into
/root",$result));
    // close session
    $session->close();
  }
```

Creating and modifying BaseX database accounts

As is custom with all database systems, for safety reasons you should **not** use the admin account for connecting to the database in your application. At the very least you should modify the admin password.
The default admin account (username/password) of BaseX is admin/admin.
To change the password you can start the BaseX standalone app (Windows Start menu >> cmd + Enter >> navigate to the BaseX bin folder >> basex.bat) , enter the “password” command and enter a new password.
Alternatively, you can use the GUI:

- Go to Database >> Server administration
- Login as admin with the default password
- If logging is successful: Go to the Users tab
- Click on the admin user, then click Alter
- Enter a new password

In the Users tab you can also add users. For programming purposes, add a user with all rights except admin, and use that account when connecting to the database from your code.
More information on User management can be found at
http://docs.basex.org/wiki/User_Management

XQuery

When I started exploring BaseX I was already familiar with XML, XSLT and XPath.
I think that XML and XPath are the minimum requirements before starting of with BaseX.
You just MUST know what XML documents look like, and how you can search them using XPath expressions.

XQuery is a step beyond XPath. It offers you a way to query a document, multiple documents or XML fragments in a very powerful way, much resembling the SQL querying language for relational databases.
I will not discuss the XQuery language here - it is way too extended to be discussed here - but the core of it is:

- make the necessary initializations
- perform a FLWOR ("flower") expression to retrieve and organize XML data
- return an XML fragment

The part you may be unfamiliar with here is FLWOR: it's pronounced "Flower" and is an acronym for ***For***, ***Let***, ***Where***, ***Order*** by and ***Return***.
The ***For*** part is like the SELECT part in a SQL query
Let lets you create subselections or values in (temporary) variables
The ***Where*** part narrows down the data you want to retrieve
The ***Order*** by part sorts the results
and the ***Return*** part can be used to format the XML fragment that you want to be returned.

My first challenge with XQuery in BaseX was: how do I tell BaseX in what database to search?
I was familiar with the doc() function of XQuery

```
for $element in doc("elements.xml")//*
```

but doc() relates to a document, not to a database.
A little Googling taught me that for databases, i.e. collections of XML documents, you can use the collection() function.
So, for example, what I wanted was something like

```
for $country in collection("Facts")//country
```

where "Facts" is a BaseX database.

JSON or no JSON

As mentioned in the introduction of this document, the reason for me to start exploring BaseX was that I wanted to compose pivot tables out of semi-structured XML data. To be more precise: I was looking for a way to store a JSON representation of XML data in BaseX, query the data with XQuery and retrieve the results as JSON.
The choice for JSON was a practical one: the application that needed the solution was a JavaScript application. XML is not native in JavaScript, while JSON is. Thus it is much more convenient to work with JSON than XML in JavaScript.
Another thing I had to keep in mind was that the JSON *coming from* BaseX – directly or indirectly - should have the same structure as the JSON *going to* BaseX.

So my first approach was to store the JSON data unmodified in BaseX, i.e. as JSON, not as XML. At that moment I was in the assumption that BaseX could store both formats, even intermingled, in its databases. Well, this appeared not to be the case. BaseX will parse incoming JSON data and convert it to XML. In the process the structure is changed, so for me that was a no-go.

My second approach was: convert the JSON data in a PHP script and store the converted data as XML in BaseX. Then retrieve the results of queries on that data directly from the BaseX web services.
Because I wanted the returned data in JSON format, I dug into the JSON serialization options of BaseX. Well, in step 5 you have read about the outcome. Again, not workable for my situation (though I must admit I did not push it to the edge. There might be a feasible solution in BaseX).
Another reason to leave this path is that an AJAX call from Javascript will not succeed if the caller and callee are not in the same domain, or have different port numbers. So, even I had found a workable JSON solution in BaseX, I would also have had to configure the BaseX HTTP server to allow cross-origin calls.

My third approach was: use PHP for both the incoming and outgoing JSON conversions. Also use PHP to query BaseX using the REST interface. Create a PHP script per query and call that from JavaScript.
This approach was successful in the sense that I had control over the JSON conversion and I could guarantee that the structure of the incoming JSON data was the same as of the outgoing data. Also this way I could circumvent the cross-origin JavaScript problem.
However, I quickly found out that one-PHP-script-per-query was not a very elegant or efficient solution. Instead I created the generic BaseXRest.php file that, essentially, can call ANY BaseX query, with the appropriate parameters where needed.
Also it seemed to me that keeping the .xq files close to BaseX was the preferred solution. No mixing up of code and content in PHP. Plus: easy testing of the .xq files in BaseX's GUI.

So, the fourth and final attempt consisted of

- Store XQueries in .xq files, in BaseX's webapp folder (easier to maintain)
- Create a generic PHP script that calls the BaseX REST service, with parameters for the .xq file to call, and optional additional parameters
- JavaScript calls PHP script – via AJAX - which delivers JSONP
- BaseX stores and delivers XML (where it's best in), PHP receives and converts to JSON and makes the JSONP