XSP Tutorial

About

E**x**tensive **S**erver **P**ath (or **XSP**) is an amalgam of XPath and XML that was written in PHP, and was inspired by XQuery and MySQL. It aims to improve the way coders use XML for storage of data by using a query-like syntax similar to MySQL to create and retrieve data. It was created by Jimmy T. Shannon Jr. in 2012 and is available as free, open-source software under the GNU Public License.  
 The language has a very basic, easy-to-understand syntax that is both very efficient and easy to learn. XSP is currently in its developing stages and right now is at a ‘beta’ build meant mainly for testing purposes. PHP must be installed on the developer’s system in order to run XSP, since that is what it was written in.  
 Usage of the language is very easy-to-understand, and can provide a lot of shortcuts that would ordinarily require a lot of code to complete. The code package comes with an XSP console (also written in PHP), that provides real-time actions and results.  
 This tutorial aims to provide a straight-forward, beginner's guide to using the new language. Please donate to help us fund this project. Documentation should be included with this web application. If not, then please contact me at [admin@zollernverse.org](mailto:admin@zollernverse.org).

**\*\*** It is highly recommended that you have some knowledge of XPath and XML before attempting XSP. **\*\***

First, we'll start off with something simple: using XSP to create an XML file.

**1.  
Creating and Deleting Files**

Creating a file is probably the easiest thing to understand in XSP. The syntax is as follows:

**create filename.xml with root(element)**

This creates the filename.xml file (you may replace filename with whatever you wish to name your file), and its root element will be, in this case, element (which is also interchangeable). Simple enough to understand, and gets to the point quickly.

**Note:** Raw XML files will not generate with an XSLT stylesheet. You must use XSP classes for that (under construction), or create and implement one yourself.

Deleting the file is even easier. You can always go through and do it manually, but XSP is nice enough to do it for you. The syntax is as follows:

**delete filename.xml**

There, simple enough, and doesn't take up a whole lot of room. So now that you know how to create and delete files, next we're going to look at storing and retrieving information from them.

Oh, wait. One more thing though that was a rather recent addition. You can now use the **fcheck** command to check if a file exists, and if it does, the parser will delete it. After this, it will recreate it. Observe:

**fcheck filename.xml with root(parentnode)**

**2.  
Retrieving and Storing Information**

Say that you want to print out the value of a specific node in an XML file. In PHP, it would take an intermediate understanding of their XML API. With XSP, however, retrieving information from an XML node couldn't be easier. The syntax is as follows:

**select //element from example.xml**

This prints the value of the “element” node, and outputs it to the client on the web page. This is NOT the equivalent to a “return” statement in programming, as it simply prints it out, and does not store the value anywhere. For this, you use the **return** command. The syntax to do this is as follows:

**return //element from example.xml**

So the syntax is pretty much the same, you're just using a different command. The difference is that the return command simply returns the value, whereas the **select** command prints it out. This command is useful for storing the value of a node into a PHP variable. It makes manipulating the files and retrieving their information for other applications much easier.

**\*\* Note:** Notice how //element is selected using XPath. Any XPath selector would work here to retrieve the data desired.

You can also select multiple elements at once:

**select //elem1,//elem2,//elem3,//elem4 from example.xml**

It works the same with the **return** statement, too. The only difference is that **select** prints out all requested data, separated by commas, whereas **return** returns all data in an associative array, for PHP to print out (similar to mysql\_fetch\_assoc() in PHP).

But what if instead of retrieving the value of a node, you wanted the value of one of its attributes? Simple, then you use the **get** command:

**get attr @name from //element in example.xml**

This prints the value of the “name” attribute from the “element” node – it does not return it. To do that, you must use the **returnattr** command:

**returnattr @name from //element in example.xml**

The syntax is a little different, but this will store the value of an attribute in an outside variable, such as PHP. This also strengthens the user-friendly aspect of XSP.

**\*\* Note:** Attribute names, like XPath, must be preceded by an ‘at’ symbol (@). This tells the parser that it’s looking for an attribute, and not an element. Alternatively, you can select attributes using the select and return commands. To get the attribute value of “lang” in “title”, you could do something like this:

**select //book[@lang=”en”]/@author from books.xml**

Attributes are also handy for another reason, as the **select** and **return** statements get even better. In contrast to the above example, you can also do this:

**select //book[1]/@lang from books.xml where //book[1]/@author = “John Boy”**

See that? It works pretty much the same as MySQL in this sense. Both examples do the same thing, but try some combinations and see what you can come up with. I haven’t tested this to its full potential just yet. Beat me to it.

You can also *set* attributes. There are two ways to do this. The first way is a single-line command. The other is a command that lets you set multiple attributes at once.

Single:

**set attr @name=”Jimmy” to //customer[1] in customers.xml**

Multiple:

**attrset @attr1=”value1”,@attr2=”value2” to //singleelement in filename.xml**

Each command has its own drawback. The first one is a lot simpler, but can only add one attribute at a time. The second can add many attributes at once, but can only do so to a single element. There is not yet a set way to affect multiple elements with different attributes in one command.

Using just these few commands together, you could write some powerful scripts in a very short time. This is what XSP is designed to do for you. If at any time you need help with the commands, remember that XSP is console-friendly. Simply type **help** to get the documentation message.

Alright, so we've looked into how to *retrieve* data, but what about how to *store* it? Well, fortunately, XSP comes with an **append** command, to add text (value) to elements, even child nodes, but for right now, we'll stick with adding text:

**append text “Hello World” to //element in filename.xml**

What the above does is add the text “Hello World” to the element tag in filename.xml. Very useful for applications such as database storage, or even just holding something temporarily. Say that you wanted to add more tags on to your file from the script, though. Maybe you've added in a new field to your program that requires something like a student's grade. Easy, simply do it like this:

**append element grade to //student in classlist.xml**

The above command generates a “grade” node underneath the student node, as a child. You can then use XSP's append command to add text to the new element as needed, as if it were any other element.

You can also now use the append command to add both the element and text at the same time:

**append element grade[“A”] to //student in classlist.xml**

And, if you want to add attributes, you can do that as well:

**append element grade[“A”]@note=”Excellent” to //student in classlist.xml**

Simply separate each attribute with another at sign (@).

You can also use XSP to replace the value in a specific node with something else. To do this, you use the **change** command:

**change nodeValue of //elem to "new value" in example.xml**

This command will change the text inside of the elem node to “new value” in the example.xml file. This command is very glitchy right now, however.  
  
**Note:** “nodeValue” does not require camel-case formatting; it will work in any casing.

Next up is the **update** command. This command is powerful in that it lets you add text and elements at the same time, and you can add multiple on one line, like so:

**update file.xml add title[“My\_Book”],author[“John\_Doe”] to /rootelement**

You can also use this not only to *add* elements, but to *change* them, as well.

**update file.xml set //title[1]=”New\_Title”,author=”Billy\_Bob”**

Seems pretty straightforward, right? XSP is always growing more powerful and capable.

You can also use XSP to delete a node and all of its child nodes, by using the **drop** command:

**drop //child[1] from example.xml**

Removes the first “child” element it finds in example.xml, along with any nodes underneath it.

**3.  
Variables & Conditions**

Variables are a fairly new addition to XSP, but they are still quite powerful. You can store values, retrieve them, change them, delete them, and do basic loops with them.

To start off with, you have the **if exists(filename.xml)** check:

**if exists(life.xml) then <out “Yay, I have one!”>  
if exists(life.xml) then <out “Yay, I have one!”> else <out “:(”>.**

This one functions exactly like it sounds. If the file exists, it prints the first message, and if it doesn’t, it prints the second. **fcheck** is like a shortcut that does both of those at once. Remember to use it where possible.

*Note that ‘out’ and ‘say’ (interchangeable commands) are simply XSP’s version of a String output (i.e., “Hello World”), but can also be used to output variables. To output both, you may concatenate them using an ampersand (****&****), though it seems only one may be used currently, due to a glitch*

~~So, now we’ve talked about the absolute basics of variables~~. But there also is *another type* of variable set, called **Complex Variables**. ~~What we have been studying so far have been~~ **~~Simplex Variables~~**. So, what’s the difference? It’s actually pretty huge: the difference between ~~Simplex~~ and Complex variables is that ~~simplex variables are stored in the library’s generic memory.xml file~~, whereas complex variables are stored within a file called variables.xml that rests inside of a ‘bin’ folder. This allows for more powerful and dynamic manipulation, callback, defining, changing and removing. This is the first step towards moving out from the memory.xml file altogether. Eventually, everything about the library will have a ‘complex’ set, while the simplex will still be somewhat in minor use. It also allows for the future of full integration of OOP, or Object Oriented Programming, in XSP.

The usage of the complex variable system is actually much simpler than that of the simplex, due to its readability and uniform functionality. For instance, read these lines, taken directly from the Documentation.txt file contained within the package:

>> **var set name = "Jimmy"**

Complex variables: creates a new variable element in bin/variables.xml, setting a name attribute to the name of the variable and a text node to the value of the variable. If the variable is already set, the XSP internal memory drops the original variable, and then re-adds it with its new value.

>> **var get name**

Complex variables: searches for and displays the value of the name variable.

>> **var return name**

Complex variables: searches for and returns the value of the name variable (does not print it out).

>> **var drop name**

>> **var delete name**

Complex variables: deletes the name variable.

>> **variable**

An alias of var.

Again, these are only the basic building blocks. Due to the nature of complex variables, they are much more powerful, and allow a lot more functionality. For instance, you could use complex variables to pull the value of an element node, and then store that value inside of a variable for XSP itself, rather than having to simply rely on PHP. It all works with the eval() method. Observe the example below:

**var set bookname = eval(return //book[1]/title from books.xml)**

Here, we are setting the bookname variable’s value like normal, up until we get to the right assignment side. At this point, we provide an XSP command to return (NOT select!) the value. It parses this first, and *then* assigns *that* value to the variable. This is one of the most powerful aspects of XSP – it can be used for many potentially powerful scripts and programs.  
 It gets even better, though. Say that you wanted to copy the value of one variable to another. It’s very simple, because all we have to do is throw in the “var get” command into the eval() method, and it will return our desired variable’s value. For example:

**var set favoritebook = eval(var return bookname)**

This declares a new variable, favoritebook, and sets its value to that of the bookname variable, because we’re returning it using the method. More powerful methods will be introduced in the future, to do more powerful functions and calculations.

Alternatively, there is an experimental method for doing this as well:

**var set favoritebook = var(bookname)**

These ‘methods’ are new to XSP, and are therefore bound to have issues…but in the meantime, they sure are useful.

**4.  
File Manipulation**

So, we’ve talked about storing, changing, retrieving and printing data – now we want to focus on further control of data: manipulation. One of the first things we’re going to look at is moving data, from one node in one file, to a different node in another file. You could also do it in the same file, depending on how the statement is written. It allows you more control over what goes where, such as if you accidentally put something somewhere it shouldn’t have gone. Otherwise, you would spend much more time digging through XML files and data (imagine a REALLY BIG file with A LOT of data). This saves you from that by simply using a command (provided of course that you know what goes where, but that’s beyond the scope of this tutorial unfortunately). The syntax for this is as follows:

**move //book from students.xml to /books in bookstore.xml**

Obviously, we don’t want our book listed as a student. That could get messy and it wouldn’t have very good grades. So, this moves it to our bookstore file. If you don’t understand the XPath statements, don’t fret – they’re very easy to learn at w3schools and you can use simple XPath selectors just as easily here as you can advanced ones, so an intermediate knowledge of it is not necessarily required.

But what if you don’t want to actually *move* the data from one element to another? What if you simply wish to copy it? Simple, then you use the **copy** command. Its syntax is exactly the same as move, verbatim; the only difference is that you type copy instead of move:

**copy //book from students.xml to /books in bookstore.xml**

It does the exact same thing as move, but *without* deleting the original value. This makes for strong, reliable mobility for files.

One of the neatest things about XSP is its ability to parse external, .xsp files. Say that you have what we have below in helloworld.xsp:

**#!xsp;  
say “Hello World!”**

Notice at the top of the file, the part that reads “#!xsp;”. This MUST be the VERY first line in your file or it will be ignored. The next part is pretty self-explanatory—we’re telling the file to print out our String. Any XSP code would work here—simply separate each section of code with a semicolon.

Now, our file by itself won’t do anything. If you load it in a browser, you *will* be able to see the code. We can run our code in the file, however, using a special command from XSP. This command is called **parse**:

**parse helloworld.xsp**

This tells the parser to load the specified file and run whatever code it finds in it, until it gets to the very last line. You could also include a ‘parse filename.xsp’ command at the top or bottom of the file, using it as a sort of makeshift ‘include’.

**\*\*NOTE:** At least 10 example.xsp files were provided for demonstration, each with varying XSP code. Note also that even though the parser requires you to type out the .xsp extension, the filename MUST end with .xsp, or it will not be parsed—this was implemented both as a security measure and a preparation for future versions.

To somewhat protect your code from prying eyes, there is a very minor thing you can do to it: you can use the **compile** command:

**compile example.xsp**

Now, the compiled version isn’t anything fancy—it’s just a blob of hexadecimal data. Any decent programmer will be able to translate it with minimal difficulty. However, this *does* provide at least *some* protection, especially from those who are unfamiliar with hex (which, admittedly, is not very likely). More advanced compilation algorithms will probably be used in the future, but right now this is what there is.

But now there’s another issue: with our code ‘compiled’, we can’t run it. This also punches a hole in the whole “security” thing. Luckily, though, we have a **decompile** command, as well:

**decompile example.xsp**

This command converts compiled XSP files back into their normal, readable state, allowing them to be parsed. But what if you wanted it to stay compiled all the time, AND run it? There is also a work-around for that. You can write an XSP file with the needed commands. For example, consider the following code, AFTER compiling our example file:

compiler.xsp:

**#!xsp;**

**#Compiler script, runs example.xsp;**

**decompile example.xsp;**

**parse example.xsp;**

**compile example.xsp**

Done. This decompiles it and runs it, then compiles and re-saves it, all without displaying its code. Now, if we run the command **parse compiler.xsp**, it will print our “Hello World” method from example.xsp, while still leaving it compiled. Neat.

Alternatively, you can use the **cparse** command to do it all in one fell swoop:

**cparse example.xsp**

Your file will need to be ‘compiled’ for this to work, but you’ll never have to ‘decompile’ it or make any fancy workarounds.

**5.  
Modules**

Modules are fairly new to XSP as well, and are in their developing stages. They are basically the language’s versions of functions. As of right now they don’t have very much use, but still come in handy sometimes. I highly advise using .xsp files for now in place of this.