NAME

XML::Simple::FAQ – Frequently Asked Questions about XML::Simple

Basics

What should I use XML::Simple for?

Nothing!

It's as simple as that.

Choose a better module. See Perl XML::LibXML by Example <http://grantm.github.io/perl-libxml-by-example/> for a gentle introduction to XML::LibXML with lots of examples.

What was XML::Simple designed to be used for?

XML::Simple is a Perl module that was originally developed as a tool for reading and writing configuration data in XML format. You could use it for other purposes that involve storing and retrieving structured data in XML but it's likely to be a frustrating experience.

Why store configuration data in XML anyway?

It seemed like a good idea at the time. Now, I use and recommend Config::General which uses a format similar to that used by the Apache web server. This is easier to read than XML while still allowing advanced concepts such as nested sections.

At the time XML::Simple was written, the advantages of using XML format for configuration data were thought to include:

- Using existing XML parsing tools requires less development time, is easier and more robust than developing your own config file parsing code
- XML can represent relationships between pieces of data, such as nesting of sections to arbitrary levels (not easily done with .INI files for example)
- XML is basically just text, so you can easily edit a config file (easier than editing a Win32 registry)
- XML provides standard solutions for handling character sets and encoding beyond basic ASCII (important for internationalization)
- If it becomes necessary to change your configuration file format, there are many tools available for performing transformations on XML files
- XML is an open standard (the world does not need more proprietary binary file formats)
- Taking the extra step of developing a DTD allows the format of configuration files to be validated before your program reads them (not directly supported by XML::Simple)
- Combining a DTD with a good XML editor can give you a GUI config editor for minimal coding effort

What isn't XML::Simple good for?

The main limitation of XML::Simple is that it does not work with 'mixed content' (see the next question). If you consider your XML files contain marked up text rather than structured data, you should probably use another module.

If your source XML documents change regularly, it's likely that you will experience intermittent failures. In particular, failure to properly use the ForceArray and KeyAttr options will produce code that works when you get a list of elements with the same name, but fails when there's only one item in the list. These types of problems can be avoided by not using XML::Simple in the first place.

If you are working with very large XML files, XML::Simple's approach of representing the whole file in memory as a 'tree' data structure may not be suitable.

What is mixed content?

Consider this example XML:

```
<document>
  <para>This is <em>mixed</em> content.</para>
</document>
```

This is said to be mixed content, because the <para> element contains both character data (text content) and nested elements.

Here's some more XML:

```
<person>
  <first_name>Joe</first_name>
  <last_name>Bloggs</last_name>
  <dob>25-April-1969</dob>
</person>
```

This second example is not generally considered to be mixed content. The <first_name>, <last_name> and <dob> elements contain only character data and the <person> element contains only nested elements. (Note: Strictly speaking, the whitespace between the nested elements is character data, but it is ignored by XML::Simple).

Why doesn't XML::Simple handle mixed content?

Because if it did, it would no longer be simple :–)

Seriously though, there are plenty of excellent modules that allow you to work with mixed content in a variety of ways. Handling mixed content correctly is not easy and by ignoring these issues, XML::Simple is able to present an API without a steep learning curve.

Which Perl modules do handle mixed content?

Every one of them except XML::Simple :–)

If you're looking for a recommendation, I'd suggest you look at the Perl-XML FAQ at:

```
http://perl-xml.sourceforge.net/faq/
```

Installation

How do I install XML::Simple?

If you're running ActiveState Perl, or Strawberry Perl <http://strawberryperl.com/> you've probably already got XML::Simple and therefore do not need to install it at all. But you probably also have XML::LibXML, which is a much better module, so just use that.

If you do need to install XML::Simple, you'll need to install an XML parser module first. Install either XML::Parser (which you may have already) or XML::SAX. If you install both, XML::SAX will be used by default.

Once you have a parser installed ...

On Unix systems, try:

```
perl -MCPAN -e 'install XML::Simple'
```

If that doesn't work, download the latest distribution from ftp://ftp.cpan.org/pub/CPAN/authors/id/G/GR/GRANTM , unpack it and run these commands:

```
perl Makefile.PL
make
make test
make install
```

On Win32, if you have a recent build of ActiveState Perl (618 or better) try this command:

```
ppm install XML::Simple
```

If that doesn't work, you really only need the Simple.pm file, so extract it from the .tar.gz file (eg: using WinZIP) and save it in the \site\lib\XML directory under your Perl installation (typically C:\Perl).

I'm trying to install XML::Simple and 'make test' fails

Is the directory where you've unpacked XML::Simple mounted from a file server using NFS, SMB or some other network file sharing? If so, that may cause errors in the following test scripts:

```
3_Storable.t
4_MemShare.t
5_MemCopy.t
```

The test suite is designed to exercise the boundary conditions of all XML::Simple's functionality and these three scripts exercise the caching functions. If XML::Simple is asked to parse a file for which it has a cached copy of a previous parse, then it compares the timestamp on the XML file with the timestamp on the cached copy. If the cached copy is *newer* then it will be used. If the cached copy is older or the same age then the file is re-parsed. The test scripts will get confused by networked filesystems if the workstation and server system clocks are not synchronised (to the second).

If you get an error in one of these three test scripts but you don't plan to use the caching options (they're not enabled by default), then go right ahead and run 'make install'. If you do plan to use caching, then try unpacking the distribution on local disk and doing the build/test there.

It's probably not a good idea to use the caching options with networked filesystems in production. If the file server's clock is ahead of the local clock, XML::Simple will re-parse files when it could have used the cached copy. However if the local clock is ahead of the file server clock and a file is changed immediately after it is cached, the old cached copy will be used.

Is one of the three test scripts (above) failing but you're not running on a network filesystem? Are you running Win32? If so, you may be seeing a bug in Win32 where writes to a file do not affect its modification timestamp.

If none of these scenarios match your situation, please confirm you're running the latest version of XML::Simple and then email the output of 'make test' to me at grantm@cpan.org

Why is XML::Simple so slow?

If you find that XML::Simple is very slow reading XML, the most likely reason is that you have XML::SAX installed but no additional SAX parser module. The XML::SAX distribution includes an XML parser written entirely in Perl. This is very portable but not very fast. For better performance install either XML::SAX::Expat or XML::LibXML.

Usage

How do I use XML::Simple?

If you don't know how to use XML::Simple then the best approach is to learn to use XML::LibXML <http://grantm.github.io/perl-libxml-by-example/> instead. Stop reading this document and use that one instead.

If you are determined to use XML::Simple, it come with copious documentation, so read that.

There are so many options, which ones do I really need to know about?

Although you can get by without using any options, you shouldn't even consider using XML::Simple in production until you know what these two options do:

- forcearray
- keyattr

The reason you really need to read about them is because the default values for these options will trip you up if you don't. Although everyone agrees that these defaults are not ideal, there is not wide agreement on what they should be changed to. The answer therefore is to read about them (see below) and select values which are right for you.

What is the forcearray option all about?

Consider this XML in a file called ./person.xml:

```
<person>
  <first_name>Joe</first_name>
  <last_name>Bloggs</last_name>
  <hobbie>bungy jumping</hobbie>
  <hobbie>sky diving</hobbie>
  <hobbie>knitting</hobbie>
</person>
```

You could read it in with this line:

```
my $person = XMLin('./person.xml');
```

Which would give you a data structure like this:

```
$person = {
  'first_name' => 'Joe',
  'last_name'  => 'Bloggs',
  'hobbie'     => [ 'bungy jumping', 'sky diving', 'knitting' ]
};
```

The <first_name> and <last_name> elements are represented as simple scalar values which you could refer to like this:

```
print "$person->{first_name} $person->{last_name}\n";
```

The <hobbie> elements are represented as an array – since there is more than one. You could refer to the first one like this:

```
print $person->{hobbie}->[0], "\n";
```

Or the whole lot like this:

```
print join(', ', @{$person->{hobbie}} ), "\n";
```

The catch is, that these last two lines of code will only work for people who have more than one hobbie. If there is only one <hobbie> element, it will be represented as a simple scalar (just like <first_name> and <last_name>). Which might lead you to write code like this:

```
if(ref($person->{hobbie})) {
  print join(', ', @{$person->{hobbie}} ), "\n";
}
else {
  print $person->{hobbie}, "\n";
}
```

Don't do that.

One alternative approach is to set the forcearray option to a true value:

```
my $person = XMLin('./person.xml', forcearray => 1);
```

Which will give you a data structure like this:

```
$person = {
  'first_name' => [ 'Joe' ],
  'last_name'  => [ 'Bloggs' ],
  'hobbie'     => [ 'bungy jumping', 'sky diving', 'knitting' ]
};
```

Then you can use this line to refer to all the list of hobbies even if there was only one:

```
print join(', ', @{$person->{hobbie}} ), "\n";
```

The downside of this approach is that the <first_name> and <last_name> elements will also always be represented as arrays even though there will never be more than one:

```
print "$person->{first_name}->[0] $person->{last_name}->[0]\n";
```

This might be OK if you change the XML to use attributes for things that will always be singular and nested elements for things that may be plural:

```
<person first_name="Jane" last_name="Bloggs">
  <hobbie>motorcycle maintenance</hobbie>
</person>
```

On the other hand, if you prefer not to use attributes, then you could specify that any <hobbie> elements should always be represented as arrays and all other nested elements should be simple scalar values unless there is more than one:

```
my $person = XMLin('./person.xml', forcearray => [ 'hobbie' ]);
```

The forcearray option accepts a list of element names which should always be forced to an array representation:

```
forcearray => [ qw(hobbie qualification childs_name) ]
```

See the XML::Simple manual page for more information.

What is the keyattr option all about?

Consider this sample XML:

```
<catalog>
  <part partnum="1842334" desc="High pressure flange" price="24.50" />
  <part partnum="9344675" desc="Threaded gasket"      price="9.25" />
  <part partnum="5634896" desc="Low voltage washer"   price="12.00" />
</catalog>
```

You could slurp it in with this code:

```
my $catalog = XMLin('./catalog.xml');
```

Which would return a data structure like this:

```
$catalog = {
    'part' => [
        {
          'partnum' => '1842334',
          'desc'    => 'High pressure flange',
          'price'   => '24.50'
        },
        {
          'partnum' => '9344675',
          'desc'    => 'Threaded gasket',
          'price'   => '9.25'
        },
        {
          'partnum' => '5634896',
          'desc'    => 'Low voltage washer',
          'price'   => '12.00'
        }
    ]
};
```

Then you could access the description of the first part in the catalog with this code:

```
print $catalog->{part}->[0]->{desc}, "\n";
```

However, if you wanted to access the description of the part with the part number of "9344675" then you'd have to code a loop like this:

```
foreach my $part (@{$catalog->{part}}) {
  if($part->{partnum} eq '9344675') {
    print $part->{desc}, "\n";
    last;
  }
}
```

The knowledge that each <part> element has a unique partnum attribute allows you to eliminate this search. You can pass this knowledge on to XML::Simple like this:

```
my $catalog = XMLin($xml, keyattr => ['partnum']);
```

Which will return a data structure like this:

```
$catalog = {
  'part' => {
    '5634896' => { 'desc' => 'Low voltage washer',   'price' => '12.00' },
    '1842334' => { 'desc' => 'High pressure flange', 'price' => '24.50' },
    '9344675' => { 'desc' => 'Threaded gasket',      'price' => '9.25'  }
  }
};
```

XML::Simple has been able to transform `$catalog->{part}` from an arrayref to a hashref (keyed on partnum). This transformation is called 'array folding'.

Through the use of array folding, you can now index directly to the description of the part you want:

```
print $catalog->{part}->{9344675}->{desc}, "\n";
```

The 'keyattr' option also enables array folding when the unique key is in a nested element rather than an attribute. eg:

```
<catalog>
  <part>
    <partnum>1842334</partnum>
    <desc>High pressure flange</desc>
    <price>24.50</price>
  </part>
  <part>
    <partnum>9344675</partnum>
    <desc>Threaded gasket</desc>
    <price>9.25</price>
  </part>
  <part>
    <partnum>5634896</partnum>
    <desc>Low voltage washer</desc>
    <price>12.00</price>
  </part>
</catalog>
```

See the XML::Simple manual page for more information.

So what's the catch with 'keyattr'?

One thing to watch out for is that you might get array folding even if you don't supply the keyattr option. The default value for this option is:

```
[ 'name', 'key', 'id']
```

Which means if your XML elements have a 'name', 'key' or 'id' attribute (or nested element) then they may get folded on those values. This means that you can take advantage of array folding simply through careful choice of attribute names. On the hand, if you really don't want array folding at all, you'll need to set 'key attr to an empty list:

```
my $ref = XMLin($xml, keyattr => []);
```

A second 'gotcha' is that array folding only works on arrays. That might seem obvious, but if there's only one record in your XML and you didn't set the 'forcearray' option then it won't be represented as an array and consequently won't get folded into a hash. The moral is that if you're using array folding, you should always turn on the forcearray option.

You probably want to be as specific as you can be too. For instance, the safest way to parse the <catalog> example above would be:

```
my $catalog = XMLin($xml, keyattr => { part => 'partnum'},
                          forcearray => ['part']);
```

By using the hashref for keyattr, you can specify that only <part> elements should be folded on the 'partnum' attribute (and that the <part> elements should not be folded on any other attribute).

By supplying a list of element names for forcearray, you're ensuring that folding will work even if there's only one <part>. You're also ensuring that if the 'partnum' unique key is supplied in a nested element then that element won't get forced to an array too.

How do I know what my data structure should look like?

The rules are fairly straightforward:

- each element gets represented as a hash
- unless it contains only text, in which case it'll be a simple scalar value
- or unless there's more than one element with the same name, in which case they'll be represented as an array
- unless you've got array folding enabled, in which case they'll be folded into a hash
- empty elements (no text contents **and** no attributes) will either be represented as an empty hash, an empty string or undef – depending on the value of the 'suppressempty' option.

If you're in any doubt, use Data::Dumper, eg:

```
use XML::Simple;
use Data::Dumper;

my $ref = XMLin($xml);

print Dumper($ref);
```

I'm getting 'Use of uninitialized value' warnings

You're probably trying to index into a non-existant hash key – try Data::Dumper.

I'm getting a 'Not an ARRAY reference' error

Something that you expect to be an array is not. The two most likely causes are that you forgot to use 'forcearray' or that the array got folded into a hash – try Data::Dumper.

I'm getting a 'No such array field' error

Something that you expect to be a hash is actually an array. Perhaps array folding failed because one element was missing the key attribute – try Data::Dumper.

I'm getting an 'Out of memory' error

Something in the data structure is not as you expect and Perl may be trying unsuccessfully to autovivify things – try Data::Dumper.

If you're already using Data::Dumper, try calling *Dumper()* immediately after *XMLin()* – ie: before you attempt to access anything in the data structure.

My element order is getting jumbled up

If you read an XML file with *XMLin()* and then write it back out with *XMLout()*, the order of the elements will likely be different. (However, if you read the file back in with *XMLin()* you'll get the same Perl data structure).

The reordering happens because XML::Simple uses hashrefs to store your data and Perl hashes do not really have any order.

It is possible that a future version of XML::Simple will use Tie::IxHash to store the data in hashrefs which do retain the order. However this will not fix all cases of element order being lost.

If your application really is sensitive to element order, don't use XML::Simple (and don't put order-sensitive values in attributes).

XML::Simple turns nested elements into attributes

If you read an XML file with *XMLin()* and then write it back out with *XMLout()*, some data which was originally stored in nested elements may end up in attributes. (However, if you read the file back in with *XMLin()* you'll get the same Perl data structure).

There are a number of ways you might handle this:

- use the 'forcearray' option with *XMLin()*
- use the 'noattr' option with *XMLout()*
- live with it
- don't use XML::Simple

Why does *XMLout()* insert <name> elements (or attributes)?

Try setting keyattr => [].

When you call *XMLin()* to read XML, the 'keyattr' option controls whether arrays get 'folded' into hashes. Similarly, when you call *XMLout()*, the 'keyattr' option controls whether hashes get 'unfolded' into arrays. As described above, 'keyattr' is enabled by default.

Why are empty elements represented as empty hashes?

An element is always represented as a hash unless it contains only text, in which case it is represented as a scalar string.

If you would prefer empty elements to be represented as empty strings or the undefined value, set the 'suppressempty' option to '' or undef respectively.

Why is ParserOpts deprecated?

The `ParserOpts` option is a remnant of the time when XML::Simple only worked with the XML::Parser API. Its value is completely ignored if you're using a SAX parser, so writing code which relied on it would bar you from taking advantage of SAX.

Even if you are using XML::Parser, it is seldom necessary to pass options to the parser object. A number of people have written to say they use this option to set XML::Parser's `ProtocolEncoding` option. Don't do that, it's wrong, Wrong, WRONG! Fix the XML document so that it's well-formed and you won't have a problem.

Having said all of that, as long as XML::Simple continues to support the XML::Parser API, this option will not be removed. There are currently no plans to remove support for the XML::Parser API.