Docs

XML Project

Derafu XML

Anonymous · 09/11/2026 · #php

Table of Contents

XML Project 3
- ***Introduction*** 4
- ***Array Structure*** 9
- ***Round Trip and Fidelity*** 19
- ***XML Document*** 23
- ***XPath Query*** 29
- ***XPath Reader*** 33
- ***XML Service*** 40
- ***Exceptions*** 46

XML Project

Derafu XML

Anonymous · 09/11/2026 · #php

Derafu XML

Last updated on 09/11/2026

Introduction

PHP library for XML to encode, decode, validate and query documents, with XPath, canonicalization and conversion from and to arrays

Anonymous · 09/11/2026

Library for XML manipulation

A PHP library to work with XML documents as data: it converts them to PHP arrays and back without losing what the XML has, validates them against XSD schemas, queries them with XPath and canonicalizes them.

Features

- **Conversion**: Transform between XML and PHP arrays in both directions. The elements, the attributes, the namespaces, the comments and the text are kept (see Round Trip and Fidelity).
- **Validation**: Validate XML documents against XSD schemas, with the errors of libxml explained and translated.
- **Querying**: XPath queries with parameters, with and without namespaces, and typed reads of the document with the same methods as `ArrayReader` (see XPath Reader).
- **Canonicalization**: Canonical XML (C14N), also in the encoding that the document declares.
- **Encoding**: The document has the encoding that it declares (UTF-8 by default, ISO-8859-1 and others).
- **Special Characters**: Automatic handling of XML special characters and entities.
- **Errors**: Exceptions with a message that can be translated, and with the details of what failed.

Installation

```
composer require derafu/xml
```

Basic Usage

Creating XML from an Array

```
use Derafu\Xml\Service\XmlEncoder;
```

```
$encoder = new XmlEncoder();

// Create an array to convert to XML.
$data = [
    'root' => [
        'element1' => 'value1',
        'element2' => 'value2',
        'element3' => [
            '@attributes' => [
                'attr1' => 'attrValue',
            ],
            '@value' => 'value3',
        ],
        'repeatedElement' => ['value4', 'value5', 'value6'],
    ],
];

// Convert the array to an XML document.
$xmlDocument = $encoder->encode($data);

// Save as XML string.
$xmlString = $xmlDocument->saveXml();

// Get the XML without the XML declaration.
$xmlContent = $xmlDocument->getXml();

// Get a canonicalized version of the XML.
$c14nXml = $xmlDocument->C14N();
```

`saveXml()` gives:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <element1>value1</element1>
  <element2>value2</element2>
  <element3 attr1="attrValue">value3</element3>
  <repeatedElement>value4</repeatedElement>
  <repeatedElement>value5</repeatedElement>
  <repeatedElement>value6</repeatedElement>
</root>
```

Converting XML to an Array

```
use Derafu\Xml\Service\XmlDecoder;
use Derafu\Xml\XmlDocument;

$decoder = new XmlDecoder();

// Load an existing XML string.
```

```
$xmlContent = '<?xml version="1.0"
encoding="UTF-8"?><root><element>value</element></root>';
$document = new XmlDocument();
$document->loadXml($xmlContent);

// Convert to an array.
$array = $decoder->decode($document);

// Now $array is ['root' => ['element' => 'value']].
```

Working with XPath

```
use Derafu\Xml\XPathQuery;

$xmlContent = '<?xml version="1.0"?><root><item id="1">First</item><item
id="2">Second</item></root>';

// Create XPath query instance.
$query = new XPathQuery($xmlContent);

// Get a specific value.
$value = $query->getValue('/root/item[@id="2"]'); // "Second"

// Get multiple values.
$values = $query->getValues('/root/item'); // ["First", "Second"]

// Get structured array.
$array = $query->get('/root');
// Result: ['item' => ['First', 'Second']]
```

Reading Typed Values

```
$reader = $document->getReader();

$reader->requireString('//seller/name');   // The text, or an error if there is none.
$reader->getAsInt('//info/number');          // The text converted to int, or null.

// One node or many is always a list, at any depth.
foreach ($reader->getReaders('//line') ?? [] as $item) {
    $item->requireString('title');
    $item->getAsNumber('qty');
}
```

See XPath Reader.

XML Validation

```
use Derafu\Xml\Exception\XmlException;
use Derafu\Xml\Service\XmlValidator;

$validator = new XmlValidator();

// Validate an XML document against a schema.
try {
    $validator->validate($xmlDocument, '/path/to/schema.xsd');
    echo "XML is valid!";
} catch (XmlException $e) {
    echo "Validation failed: " . $e->getMessage();
    $errors = $e->getErrors(); // The errors of libxml, as XmlError.
}
```

Advanced Usage

Working with Namespaces

The namespaces that an element declares are written in its `@attributes`, and the elements and attributes that use them are in them, in the document that the encoder gives:

```
$data = [
    'Invoice' => [
        '@attributes' => [
            'xmlns' => 'urn:example:invoice',
            'xmlns:cbc' => 'urn:example:basic',
        ],
        'cbc:ID' => 'F-1',
        'cbc:Note' => ['a', 'b'],
    ],
];

$document = $encoder->encode($data);

// Query with the namespaces, with no need to write the XML and load it again.
$namespaces = ['i' => 'urn:example:invoice', 'cbc' => 'urn:example:basic'];
$query = new XPathQuery($document, $namespaces);
$id = $query->get('//cbc:ID'); // "F-1"
$notes = $query->get('//cbc:Note'); // ["a", "b"]
```

Array Structure for XML Creation

When creating XML from arrays, the structure follows these conventions:

- Simple key-value pairs become element nodes.
- Use `@attributes` for node attributes and for the declarations of namespaces.

- Use `@value` for the text of a node that also has attributes, child elements or comments.
- Use `@comment` for comments.
- Arrays of values create repeated nodes with the same name.

See Array Structure for all the rules.

What Is Next

- Array Structure: how an XML is an array, and the other way around.
- Round Trip and Fidelity: what is kept when an XML becomes an array and an array becomes an XML.
- XML Document: the `XmlDocument` class.
- XPath Query: querying documents.
- XPath Reader: reading documents with types.
- XML Service: encoding, decoding and validating together.
- Exceptions: what is thrown, when, and how to translate the messages.

Last updated on 09/11/2026

Array Structure

XML-Array Conversion Structure

Anonymous · 09/11/2026

XML-Array Conversion Structure

The Derafu XML library uses a specific convention for representing the structure of XML documents as PHP arrays and vice versa. Understanding this structure is essential for effectively using the encoding and decoding features. For what is kept when an XML becomes an array and the array becomes an XML again, see Round Trip and Fidelity.

Basic Structure

At its most basic level, an XML element with a value is represented as a key-value pair in the array:

XML:

```
<element>value</element>
```

Array:

```
['element' => 'value']
```

Nested Elements

Nested elements are represented as nested arrays:

XML:

```
<parent>
  <child>value</child>
</parent>
```

Array:

```
[
    'parent' => [
        'child' => 'value'
    ]
```

```
]
```

Attributes

XML attributes are handled using the special `@attributes` key:

XML:

```
<element id="123" type="example">value</element>
```

Array:

```
[
    'element' => [
        '@attributes' => [
            'id' => '123',
            'type' => 'example'
        ],
        '@value' => 'value'
    ]
]
```

Note the use of `@value` to hold the element's text content when the element also has attributes (or child elements or comments).

Repeated Elements

Elements with the same name are collected into arrays:

XML:

```
<parent>
  <child>value1</child>
  <child>value2</child>
  <child>value3</child>
</parent>
```

Array:

```
[
    'parent' => [
        'child' => [
            'value1',
            'value2',
```

```
            'value3'
        ]
    ]
]
```

Complex Structures

The conventions can be combined to represent complex XML structures:

XML:

```
<root>
  <items>
    <item id="1">
      <name>Item 1</name>
      <price currency="USD">19.99</price>
    </item>
    <item id="2">
      <name>Item 2</name>
      <price currency="EUR">29.99</price>
    </item>
  </items>
  <summary total="2">Multiple items</summary>
</root>
```

Array:

```
[
    'root' => [
        'items' => [
            'item' => [
                [
                    '@attributes' => ['id' => '1'],
                    'name' => 'Item 1',
                    'price' => [
                        '@attributes' => ['currency' => 'USD'],
                        '@value' => '19.99'
                    ]
                ],
                [
                    '@attributes' => ['id' => '2'],
                    'name' => 'Item 2',
                    'price' => [
                        '@attributes' => ['currency' => 'EUR'],
                        '@value' => '29.99'
                    ]
                ]
```

```
            ]
        ],
        'summary' => [
            '@attributes' => ['total' => '2'],
            '@value' => 'Multiple items'
        ]
    ]
]
```

Text

The text of an element is exactly the one that the XML has, with its spaces. The library does not trim it: what to do with the spaces is for whoever uses the value.

XML:

```
<name>  Acme Inc.  </name>
```

Array:

```
['name' => '  Acme Inc.  ']
```

CDATA sections and entities are the text that they are (`<![CDATA[x < y]]>` is `'x < y'`). The spaces and the line breaks that only format the XML (the ones between elements) are not text.

When an element has child elements or comments and also text, the text is in parts, and the parts are joined with a space, without the spaces around each part (see What the array does not keep).

Namespaces

The namespaces that an element declares are in its `@attributes`, as they are written in the XML: `xmlns` for the default one and `xmlns:prefix` for the others. They go first, sorted by name, and before the other attributes. A namespace is only where it is declared, not in the elements that inherit it.

XML:

```
<Invoice xmlns="urn:example:invoice" xmlns:cbc="urn:example:basic" id="1">
  <cbc:ID>F-1</cbc:ID>
</Invoice>
```

Array:

```
[
```

```
    'Invoice' => [
        '@attributes' => [
            'xmlns' => 'urn:example:invoice',
            'xmlns:cbc' => 'urn:example:basic',
            'id' => '1',
        ],
        'cbc:ID' => 'F-1',
    ]
]
```

The names of elements and attributes keep their prefix (`cbc:ID`, `xsi:schemaLocation`). An element that leaves the default namespace without URI has `'xmlns' => ''`. With the namespaces in the array, the XML that is created from it has the same elements in the same namespaces.

Comments

The comments of an element are in its `@comment` key, as they are written (with their spaces): a text if there is one, a list of texts if there are several.

XML:

```
<element>
  <!-- first -->
  <child>value</child>
</element>
```

Array:

```
[
    'element' => [
        '@comment' => ' first ',
        'child' => 'value',
    ]
]
```

Special Cases

Empty Elements

An element without content is an empty string, whether it is written `<element/>` or `<element></element>`: it is the value of an empty element in the data model of XML.

XML:

```
<element></element>
```

Array:

```
['element' => '']
```

When creating XML, the same empty string creates an empty element. An element is a `null` only when it is skipped (see below).

Skipping Elements

When creating XML, certain values cause an element to be skipped:

```
[
    'included' => 'This will be in the XML',
    'excluded' => null,       // This will be skipped.
    'alsoExcluded' => false,  // This will be skipped.
    'emptyArraySkipped' => [] // This will be skipped.
]
```

Array to XML Encoding Rules

When using `XmlEncoder` to create XML from arrays, these rules apply:

1. Simple key-value pairs become elements with text content.
2. Nested arrays become nested elements.
3. The special key `@attributes` defines attributes for the parent element. The namespaces (`xmlns`, `xmlns:prefix`) are declared the same way.
4. The special key `@value` defines the text content of the parent element. It is written before the child elements and the comments, wherever the key is in the array.
5. The special key `@comment` adds a comment (or a list of them) to the parent element.
6. Arrays of values create multiple elements with the same name.
7. Null, false, and empty arrays are skipped (not included in the XML).
8. Empty strings (’’) create empty elements.
9. The array has exactly one element at its first level: it is the root element of the XML.
10. A value is written as text if it is a string, a number, a boolean or an object that can be a string. Anything else is not valid.
11. The names of elements and attributes are text. A number is the position of a value in a list, not a name.

What is not valid (a second root element, an attribute that is an array, a name that is not text, a value that can not be written as text, a text outside of an element, a comment that has two hyphens in a row or ends with a hyphen) throws an `XmlEncoderException`.

The XML has the encoding of the document that is returned: it is UTF-8 unless it is changed with `setEncoding()`.

The document that is returned has the namespaces that the array declares: its elements and its attributes are in them, as in a document that is loaded from the XML. XPath queries with namespaces and `C14N()` work in it without writing the XML and loading it again. A prefix that the array does not declare is a name without namespace.

XML to Array Decoding Rules

When using `XmlDecoder` to convert XML to arrays, these rules apply:

1. Elements become keys in the array.
2. Text content becomes the value of the key.
3. Attributes are collected under the `@attributes` key, with the namespaces that the element declares.
4. When the element has attributes, or child elements or comments, its text content is stored under the `@value` key.
5. Multiple elements with the same name are collected into arrays.
6. Empty elements become empty strings.
7. Comments are collected under the `@comment` key.
8. Text is exactly the one of the XML, with its spaces.
9. Complex nested structures are preserved in the array hierarchy.

What the Array Does Not Keep

An array with names as keys can not say some things that an XML can. The array that is decoded from an XML has its data, and what follows is the part that does not come back when it is encoded again:

- **The order among child elements with different names, when one of them is repeated.** The repeated elements are a list under their name.

  ```
  <r><a>1</a><b>2</b><a>3</a></r>   <!-- ['r' => ['a' => ['1', '3'], 'b' => '2']] -
  ->
  ```

  is encoded as `<r><a>1</a><a>3</a><b>2</b></r>`.

- **The place of the text and of the comments among the child elements.** The text that is mixed with elements or comments is joined in `@value`, and the comments of an element are together, in the place of the first one.

```
<p>Hello <b>big</b> world</p>   <!-- ['p' => ['@value' => 'Hello world', 'b' =>
'big']] -->
```

is encoded as `<p>Hello world<b>big</b></p>`. It is a structure for data. It is not for documents that mix text and markup, as HTML does.

- **What is outside of the root element:** comments, processing instructions (`<?xml-stylesheet ... ?>`) and the document type.
- **The processing instructions inside of the elements.**
- **The declaration of the XML:** the version and `standalone`. The encoding is the one of the document (`getEncoding()` and `setEncoding()`), not of the array.
- **The entities that a document type defines.** A reference to one of them has no text in the array.
- **How the text was written:** CDATA sections, character references (`é`), the quotes of the attributes, `<a/>` or `<a></a>`, and the spaces that format the XML.

Usage Examples

Creating Complex XML

```
use Derafu\Xml\Service\XmlEncoder;

$data = [
    'invoice' => [
        '@attributes' => [
            'id' => 'INV-2025-001',
            'date' => '2025-03-05'
        ],
        'customer' => [
            'name' => 'Acme Inc.',
            'address' => [
                'street' => '123 Main St',
                'city' => 'Anytown',
                'zipcode' => '12345'
            ]
        ],
        'items' => [
            'item' => [
```

```
                [
                    '@attributes' => ['sku' => 'PROD1'],
                    'description' => 'Product One',
                    'quantity' => '2',
                    'price' => '10.99'
                ],
                [
                    '@attributes' => ['sku' => 'PROD2'],
                    'description' => 'Product Two',
                    'quantity' => '1',
                    'price' => '24.99'
                ]
            ]
        ],
        'total' => '46.97'
    ]
];

$encoder = new XmlEncoder();

$xmlDocument = $encoder->encode($data);
echo $xmlDocument->saveXml();
```

Processing XML into Arrays

```
use Derafu\Xml\Service\XmlDecoder;

$xmlString = '
<catalog>
    <book id="bk101">
        <author>Gambardella, Matthew</author>
        <title>XML Developer\'s Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2025-01-15</publish_date>
    </book>
    <book id="bk102">
        <author>Ralls, Kim</author>
        <title>Midnight Rain</title>
        <genre>Fantasy</genre>
        <price>5.95</price>
        <publish_date>2025-02-20</publish_date>
    </book>
</catalog>';

$document = new \Derafu\Xml\XmlDocument();
$document->loadXml($xmlString);

$decoder = new XmlDecoder();
```

```
$array = $decoder->decode($document);

// Access data directly.
$firstBookTitle = $array['catalog']['book'][0]['title']; // "XML Developer's Guide"
$secondBookPrice = $array['catalog']['book'][1]['price']; // "5.95"
```

Tips for Working with the Array Structure

1. **Plan Your Structure**: When creating XML, plan your array structure carefully to match the desired XML output.
2. **Handle Repeated Elements**: Always use numeric arrays for repeated elements with the same name.
3. **Use `@attributes` and `@value`**: Remember to use these special keys when working with elements that have both attributes and text content.
4. **Validate Your Output**: Always validate the generated XML against your schema if one exists.
5. **Check for Empty Values**: When processing arrays from XML, an empty element is an empty string (`''`), not `null`.
6. **Clean the Text Where You Use It**: The text is not trimmed. If a value must have no spaces around it, trim it where it is used.

Last updated on 09/11/2026

Round Trip and Fidelity

What is kept when an XML becomes an array and an array becomes an XML, how it is tested and what to consider with other XML formats

Anonymous · 09/11/2026

Round Trip and Fidelity

The library converts XML to arrays and arrays to XML. A round trip is to do both: an XML becomes an array, and the array becomes an XML again. This page says what the XML that comes back has, how it is tested and what to consider when the XML is not the one of a document that the library was made for.

What Is Kept

When an XML is decoded and the array is encoded again, the XML that comes back has:

- The same elements, in the same namespaces, with the same names and prefixes.
- The same attributes, with their values and their namespaces (`xsi:schemaLocation` is not `schemaLocation`).
- The declarations of namespaces (`xmlns` and `xmlns:prefix`).
- The same text, exactly as it is: with its spaces, its line breaks and its special characters. The library does not trim it.
- The same comments inside of the document element.
- The same order of the child elements, when the elements that are repeated are together (what an XML schema asks for, in a sequence).
- The elements without content, as empty elements.

What is kept is the data of the XML, not the way it is written. The spaces that format the XML (the ones between elements), the quotes of the attributes, `<a/>` or `<a></a>` and the CDATA sections are not kept as they were written: they are the same data.

```
$document = new XmlDocument();
$document->loadXml($xml);

$array = $decoder->decode($document);
$result = $encoder->encode($array);

// What the XML has is the same: for example, its canonical XML.
// (Without the spaces that only format the XML.)
```

What Is Not Kept

An array of names can not say some things that an XML can: the order among child elements with different names when one of them is repeated, the place of the text and of the comments among the child elements, what is outside of the document element, the declaration of the XML and the entities that a document type defines. See What the Array Does Not Keep.

The Encoding

The encoding is not in the array: it is a property of the document that is written. Whoever uses the decoder and the encoder keeps it, with `getEncoding()` and `setEncoding()`:

```
$document = new XmlDocument();
$document->loadXml($xml); // For example, an XML that declares ISO-8859-1.
$encoding = $document->getEncoding();

$result = $encoder->encode($decoder->decode($document));
$result->setEncoding($encoding);

$result->saveXml(); // In ISO-8859-1. What it can not write is a character reference.
```

The Document That the Encoder Gives

The document that `XmlEncoder::encode()` gives has the namespaces that the array declares: its elements and its attributes are in them, as in a document that is loaded from the XML. So XPath queries with namespaces and `C14N()` work in it without writing the XML and loading it again.

How It Is Tested

The test suite makes the round trip, in both directions, with the XML of its fixtures (an example of an electronic tax document and documents of the UBL standard of versions 2.1 to 2.5) and with small XML that have what other XML have: namespaces declared in several places, attributes with prefix, comments, text with spaces, CDATA, entities, unicode and ISO-8859-1.

- **XML to array to XML**: the XML that comes back has the same elements, namespaces, attributes, text, comments and order. For the fixtures it is also the same canonical XML (C14N), compared with a method that does not depend on how the comparison above is made.
- **Array to XML to array**: the array that comes back is the same.
- **Stability**: one more round trip gives the same XML.
- **Encoder**: a test with arrays made at random checks that what is encoded is decoded as it was, that what means nothing (`null`, `false` and `[]`) does not change the XML, and that a number or `true` is the same as its text.

Using It with Other XML Formats

The library does not know the schema of the XML. It converts any XML that is data (elements, attributes and text) and keeps it, so it works with formats as UBL. What follows is what to consider when the array is used to read or to write.

One Element or a List

An element that appears once is its value. An element that is repeated is a list. The same element can be one or the other in different documents:

```
<Invoice><InvoiceLine>...</InvoiceLine></Invoice>              <!-- ['Invoice' =>
['InvoiceLine' => [...]]] -->
<Invoice><InvoiceLine>...</InvoiceLine><InvoiceLine>...</InvoiceLine></Invoice>
                                                               <!-- ['Invoice' =>
['InvoiceLine' => [[...], [...]]]] -->
```

Whoever reads an array has to consider both. If the schema says that an element can be repeated, a function that always gives a list makes the code simpler:

```
function asList(mixed $value): array
{
    return is_array($value) && array_is_list($value) ? $value : [$value];
}

foreach (asList($array['Invoice']['cac:InvoiceLine'] ?? []) as $line) {
    // ...
}
```

The Prefixes Are in the Keys

The names of the keys are the names of the XML, with the prefix that the document uses (`cbc:ID`). Two documents with the same data and other prefixes give arrays with other keys. To read a document that is not under your control, query it with XPath and the namespaces (see XPath Query), which do not depend on the prefixes.

Everything Is Text

The values are text: `'1.50'` is not a number, and a date is not a date. A layer that knows the schema converts the types.

The Order of the Array Is the Order of the XML

The encoder writes the elements in the order of the array. When an XML has to follow the order of a

schema, the array has to have the keys in that order.

Last updated on 09/11/2026

XML Document

The XmlDocument class, a document with its encoding, namespaces, canonicalization, XPath queries and conversion to an array

Anonymous · 09/11/2026

XmlDocument Class

`XmlDocument` is the XML document of the library. It has a PHP `DOMDocument` inside (`getDomDocument()` gives it), and adds what is needed to work with XML as data: the encoding of the document, access to its metadata, serialization that handles special characters, canonicalization, XPath queries and conversion to an array.

It is the document that `XmlEncoder::encode()` gives, the one that `XmlDecoder::decode()` and `XmlValidator::validate()` receive, and the one that the exceptions carry.

Key Features

- Loading with the errors of libxml in the exception.
- The encoding of the document, to read and to change.
- Access to the metadata of the document: the name of the document element, its namespace and its schema.
- Serialization with the special characters and entities handled.
- Canonical XML (C14N), also in the encoding of the document.
- XPath queries with parameters.
- Conversion to a simple array, and to JSON.

Basic Usage

Creating a New Document

```
use Derafu\Xml\XmlDocument;

// A document that is empty: UTF-8 and version 1.0.
$document = new XmlDocument();

// Or with another encoding (the first parameter) and version (the second one).
$document = new XmlDocument('ISO-8859-1', '1.0');
```

The encoding and the version are the ones of the XML declaration when the document is saved. A

document that is loaded has the ones that the XML declares.

The document does not format the XML that it saves: it only formats it if it is asked to. A document that is loaded is saved as it was loaded, with the spaces that it has between its elements, and a document that is built element by element is saved in one line. The document that `XmlService::encode()` gives is asked to format the XML (see XmlService).

```
// Save with line breaks and indentation, in the places where the document has no
// spaces between its elements. It does not matter if it is called before or after
loading.
$document->setFormatOutput(true);

// Before loading: take out the spaces between elements of the XML that is loaded.
$document->setPreserveWhiteSpace(false);
```

The spaces that an XML that is loaded already has between its elements are kept, unless `setPreserveWhiteSpace(false)` is called before `loadXml()`. They are part of the content of the document, so what is saved with `saveXml()` or `getXml()` has them, with or without `setFormatOutput(true)`: the format is only added where there are no spaces.

If the XML is signed, it has to be saved as it was loaded: the spaces between its elements are part of what is signed, so a document that is saved with another format does not have a valid signature.

Loading XML Content

```
// Load from a string.
$xmlString = '<root><element>value</element></root>';
$document->loadXml($xmlString);
```

`loadXml()` returns `true`. If the string is empty or the XML is not well formed it throws an `XmlParseException` that has the errors of libxml (see Exceptions):

```
use Derafu\Xml\Exception\XmlParseException;

try {
    $document->loadXml('<root><element></root>');
} catch (XmlParseException $e) {
    echo $e->getMessage(); // "Failed to load the XML: Line 1, column 23: Opening and
ending tag mismatch: element (line 1) and root"
    $errors = $e->getErrors(); // One XmlError for each error of libxml.
}
```

The loading does not convert the encoding: the document has the one that the XML declares, and `getEncoding()` says it (in upper case; `UTF-8` if the XML does not declare one). External entities are not loaded.

Accessing Document Information

```
// The name of the document element, with its prefix if it has one.
$name = $document->getName(); // e.g., "root"

// The default namespace of the document element (`xmlns`), if it has one.
$namespace = $document->getNamespace(); // e.g., "urn:example:doc"

// The path of the schema in xsi:schemaLocation, if there is one.
$schema = $document->getSchema(); // e.g., "/path/to/schema.xsd"

// The element itself, to work with it as a DOMElement.
$element = $document->getDocumentElement(); // null if there is no content.

// The same element, for who needs it: there is no null to check.
$element = $document->requireDocumentElement(); // XmlException if there is no
content.
```

`requireDocumentElement()` is the one to use when the code can not go on without the element. A document without content is not a value that is not there, it is a mistake of whoever asks, so it is an `XmlException` and not a `null` that would have to be checked before using it. `getName()`, `getNamespace()` and `getSchema()` use it: they throw an `XmlException` if the document has no document element (it has no content yet), because there is nothing to ask.

For `xsi:schemaLocation="urn:example:doc /path/to/schema.xsd"`, `getSchema()` gives the second part: `/path/to/schema.xsd`.

The Encoding

```
$encoding = $document->getEncoding(); // e.g., "ISO-8859-1"

// Change the encoding of the XML that is saved.
$document->setEncoding('ISO-8859-1');
```

The characters that the encoding can not write are saved as character references (`€`).

Saving and Serializing

```
// The complete XML document with the declaration.
$xmlString = $document->saveXml();

// The XML without the declaration and without initial or final line breaks.
$xmlContent = $document->getXml();
```

Both handle the special characters of the text: `&`, `<` and `>` are escaped, and the quotes are written as `"` and `'`.

```
$document->loadXml('<root><element>Text with &amp; &lt; &gt; " \'</element></root>');
$document->getXml();
// <root>
//   <element>Text with &amp; &lt; &gt; &quot; &apos;</element>
// </root>
```

Canonical XML

```
// Canonical XML (C14N). It is always in UTF-8.
$canonical = $document->C14N();

// The same, in the encoding that the document declares (UTF-8 if it has none).
$encoded = $document->C14NEncoded();

// Only a part of the document, the first node that an XPath finds.
$part = $document->C14NEncoded('/root/section');

// The same, without the spaces between the tags.
$flat = $document->C14NEncodedFlattened();
$flatPart = $document->C14NEncodedFlattened('/root/section');
```

- `C14N()` takes the parameters of the `DOMDocument::C14N()` method of PHP (exclusive, with comments, list of nodes and namespace prefixes).
- `C14NEncoded()` and `C14NEncodedFlattened()` throw an `XmlQueryException` if the XPath finds no node.
- If the XML can not be canonicalized, `C14N()`, `C14NEncoded()`, `C14NEncodedFlattened()` and `getTopLevelSignatureC14N()` throw an `XmlException`, with what libxml said in `getErrors()`. It happens, for example, when a namespace is a relative URI (`<a xmlns="relative">`): libxml loads it, but the Canonical XML recommendation requires the canonicalization to fail. PHP gives `false` for it; the library does not.

To canonicalize a node, and not the whole document, `XmlHelper::canonicalize()` takes the same parameters, with the node (or the `DOMDocument`) first, and fails in the same way:

```
use Derafu\Xml\XmlHelper;

$node = $document->getNodes('/root/section')->item(0);
$canonical = XmlHelper::canonicalize($node, exclusive: true);
```

XPath Querying

The document has XPath queries with the namespaces ignored (see XPath Query for the details and to query with namespaces):

```
// The result as a string, an array or null.
```

```
$result = $document->query('/root/element');

// The result as a DOMNodeList.
$nodes = $document->getNodes('/root/element');

// With parameters, quoted for you.
$result = $document->query('/root/element[@id=:id]', ['id' => '123']);
```

To read the document with types there is `getReader()`, which gives an `XPathReader`. It takes the namespaces of the XPath expressions and where the document comes from, for the errors:

```
$reader = $document->getReader();

$reader->requireString('/root/element');
$reader->getAsInt('/root/count');
$reader->getReaders('/root/item');

$reader = $document->getReader(['cbc' => 'urn:example:basic'], 'invoice.xml');
```

Array Conversion

`toArray()` is the simple version of the conversion: the value of each element by its name, without the attributes, the namespaces nor the comments. A document without content is an empty array. To have all of the XML, to create it again from the array for example, use the decoder (see Array Structure).

```
// Convert the entire document to an array.
$array = $document->toArray();

// Access a specific part using dot notation.
$value = $document->get('root.element');

// With a default value if not found.
$value = $document->get('root.missing', 'default value');

// The document is the array in JSON.
$json = json_encode($document);
```

Working with Signatures

If the document has a `Signature` element as a child of its document element, you can get its canonicalized XML (C14N):

```
// Get the canonicalized XML of the Signature that is a child of the document element.
```

```
$signatureXml = $document->getTopLevelSignatureC14N();

// Returns null if the document element has no Signature as a child.
if ($signatureXml !== null) {
    // Process signature...
}
```

The `Signature` is found by its local name, in any namespace or prefix, and only among the children of the document element. The result is the one that C14N gives (in UTF-8, whatever the encoding of the document is), with the declarations of the namespaces that are in scope in the `Signature`. For a `Signature` that is deeper in the document, for example inside an extension, use `C14NEncoded()` with its XPath.

Serializing the Document

The document can be serialized with `serialize()` and restored with `unserialize()`: its XML is what is kept, so it comes back with the same content and encoding, and it works the same.

```
$copy = unserialize(serialize($document));
```

Performance Considerations

- With large XML documents, prefer XPath queries to extract the part that you need rather than converting the entire document to an array.
- The canonicalization methods do more work: use them when you need the canonical XML.
- `getXml()` is `saveXml()` without the declaration, so it does a little more work, not less.

Last updated on 09/11/2026

XPath Query

The XPathQuery class, XPath queries with parameters, with and without namespaces, and what each method returns

Anonymous · 09/11/2026

XPathQuery Class

`XPathQuery` extracts data from XML documents using XPath expressions. It has methods for a value, for a list of values, for a structure and for the DOM nodes, parameters that are quoted for you, context nodes and support for namespaces.

Basic Usage

```
use Derafu\Xml\XPathQuery;

// Initialize with an XML string, an XmlDocument or a DOMDocument.
$query = new XPathQuery($xmlString);

// Get a single value.
$value = $query->getValue('/root/element');

// Get multiple values.
$values = $query->getValues('/root/items/item');

// Get DOM nodes.
$nodes = $query->getNodes('/root/items/item');

// Get structured data.
$data = $query->get('/root');
```

An XML string that is not well formed throws an `XmlParseException` when the query is created. A query that fails throws an `XmlQueryException` (see Exceptions).

What Each Method Returns

Method	Returns
`getValue($query)`	The value of the first node, as a string, or `null` if there is none.
`getValues($query)`	A list with the value of each node. A node without value is `null`.
`getNodes($query)`	A `DOMNodeList` with the nodes.

Method	Returns
`get($query)`	The structure of what the query finds, as the next section says.

The Structure That `get()` Returns

1. `null` if no node is found.
2. A string if a node that has only text is found (an element without content is an empty string; an attribute is its value).
3. An array if the node has child elements: the value of each child by its name, and a list for the elements that are repeated.
4. A list of those, if several nodes are found.

It is the simple structure of the document: the attributes, the namespaces and the comments are not in it. For all of the XML use the decoder (see Array Structure).

For this XML:

```
<root>
  <person>
    <name>John</name>
    <age>30</age>
  </person>
  <person>
    <name>Jane</name>
    <age>25</age>
  </person>
</root>
```

The query `$query->get('/root/person')` returns:

```
[
    ['name' => 'John', 'age' => '30'],
    ['name' => 'Jane', 'age' => '25'],
]
```

And `$query->get('/root/person[1]/name')` returns `'John'`.

Parameterized Queries

The values that go in a query are given as parameters: in the query the parameter is written `:name`, and in the array of parameters it is `name`. They are quoted for you, also when they have quotes:

```
$result = $query->get('//item[@id=:id and @type=:type]', [
    'id' => '123',
```

```
    'type' => 'product',
]);

$params = ['complex' => 'Value with "double" and \'single\' quotes'];
$result = $query->get('//element[text()=:complex]', $params);
```

Context Nodes

`get()`, `getValue()`, `getValues()` and `getNodes()` take a context node as the last parameter, to query from it:

```
// Get a context node.
$contextNode = $query->getNodes('//section')->item(0);

// Query within that context.
$result = $query->get('./item', [], $contextNode);
```

Working with Namespaces

By default the namespaces are ignored: the names in the query match the elements by their local name, whatever their namespace and prefix.

```
$xml = '<r xmlns="urn:d" xmlns:p="urn:p"><p:a>1</p:a><b>2</b></r>';

$query = new XPathQuery($xml);
$query->get('//a'); // "1": the element p:a, found by its name.
$query->get('//b'); // "2"
```

Only the names of the elements are read this way. The rest of the expression is what XPath says: the functions and the node tests (`count()`, `text()`), the axes (`following-sibling::`), the attributes, the operators (`and`, `or`, `div`, `mod`) and the string literals.

```
$xml = '<r xmlns="urn:d"><a>1</a><b-c id="5">2</b-c></r>';

$query = new XPathQuery($xml);
$query->get('//a/text()');                // "1"
$query->get('//b-c[@id="5"]');            // "2": a name can have a hyphen.
$query->get('//a/following-sibling::b-c'); // "2"
```

With the namespaces of the query as the second parameter they are used as the XPath says: the names have the prefix of the namespace that they are in.

```
$namespaces = ['d' => 'urn:d', 'p' => 'urn:p'];
```

```
$query = new XPathQuery($xml, $namespaces);

$query->get('//p:a'); // "1"
$query->get('//d:b'); // "2"
$query->get('//b');   // null: b is in the namespace urn:d, it is not a name without
namespace.
```

The prefixes are the ones that you give, not the ones that the document uses: they can be different. It is the way to query a document whose prefixes you do not control.

Without registering the namespaces a name with a prefix (`//p:a`) is not valid.

Querying an XmlDocument

An `XmlDocument` can be given instead of a string. Also the one that the encoder gives:

```
$document = $encoder->encode($data);

$query = new XPathQuery($document, ['cbc' => 'urn:example:basic']);
$query->get('//cbc:ID');
```

`XmlDocument::query()` and `getNodes()` use `XPathQuery` without namespaces, and they read the document as it was loaded last.

An `XPathQuery` reads the content that the document had when the query was created: if the XML is loaded again in the document, create a new query. To read a document with types, and as it is each time it is asked, see XPath Reader.

Error Handling

```
use Derafu\Xml\Exception\XmlQueryException;

try {
    $result = $query->get('//invalid[xpath');
} catch (XmlQueryException $e) {
    echo $e->getMessage();
    // An error occurred while executing the XPath expression: ... Invalid predicate.
}
```

Last updated on 09/11/2026

XPath Reader

The XPathReader class, typed reads of an XML document selected by XPath, with the same methods as ArrayReader

Anonymous · 09/11/2026

XPathReader Class

`XPathReader` reads an XML document with types. It is to an XML document what `ArrayReader` is to an array: the same methods, with the same contract, and an XPath expression where `ArrayReader` takes a key.

```
use Derafu\Xml\XmlDocument;

$document = new XmlDocument();
$document->loadXml(<<<XML
<order>
    <header>
        <info>
            <docType>33</docType>
            <number>0021</number>
            <issued>2025-03-07</issued>
        </info>
        <seller>
            <taxId>TAX-1</taxId>
            <name>Seller name</name>
            <phone>111</phone>
            <phone>222</phone>
        </seller>
    </header>
    <line>
        <title>Item A</title>
        <qty>2</qty>
        <code><value>111</value></code>
        <code><value>222</value></code>
    </line>
    <line>
        <title>Item B</title>
        <qty>3</qty>
        <code><value>333</value></code>
    </line>
    <shipping>
        <customs/>
    </shipping>
</order>
```

```
XML);

$reader = $document->getReader();

$reader->requireString('//seller/name');   // 'Seller name'
$reader->getAsInt('//info/number');          // 21
$reader->getString('//info/number');         // '0021'
$reader->getString('//seller/branch');     // null: no node matches.
$reader->requireString('//seller/branch'); // throws XmlQueryException.
```

Getting a Reader

```
use Derafu\Xml\XPathReader;

$reader = $document->getReader();
$reader = XPathReader::fromDocument($document);

// With the namespaces that the XPath expressions use, and where the document
// comes from, which the errors say.
$reader = $document->getReader(['cbc' => 'urn:example:basic'], 'invoice.xml');
```

Without namespaces they are ignored, as in `XmlDocument::query()`: the names in the XPath match the elements by their local name. With them the XPath is used as it is written, with the prefixes that you give (see XPath Query).

The reader reads the document each time it is asked, so it sees the changes that are made to the document and the XML that is loaded in it again.

Strict and Converting Reads

The text of an XML node is always a string. The reader has two kinds of reads, as `ArrayReader` does:

- `getXxx()` is strict: it returns `null` when no node matches, and throws when what it finds is not of the type. `requireXxx()` is the same, and it also throws when no node matches.
- `getAsXxx()` converts the text to the type, with the conversions of Value: `'0021'` is `21`. `requireAsXxx()` throws when no node matches.

There are no strict getters for `int`, `float`, `number` or `bool`: they could never succeed with the text of a node.

Method	Returns
`getString()`, `requireString()`	`string`, the text as it is
`getNonEmptyString()`, `requireNonEmptyString()`	`non-empty-string`; an empty element throws

Method	Returns
`getAsString()`, `requireAsString()`	`string`
`getAsInt()`, `requireAsInt()`	`int`
`getAsFloat()`, `requireAsFloat()`	`float`
`getAsNumber()`, `requireAsNumber()`	`int` or `float`, as the text is
`getAsBool()`, `requireAsBool()`	`bool`; a text that is not a boolean throws
`getDateTime()`, `requireDateTime()`	`DateTimeInterface`, from a text that PHP reads as a date (see `ValueType::dateTime()`); another text throws
`getEnum()`, `requireEnum()`	the case of a backed enum with that value; another text throws

```
$reader->getString('//info/number');       // '0021': the text, as it is.
$reader->getAsInt('//info/number');        // 21
$reader->getAsFloat('//info/number');      // 21.0
$reader->getAsBool('//info/number');       // throws: it is not a boolean text.
```

The `getXxx()` methods take a default when no node matches, as `ArrayReader` does:

```
$reader->getString('//seller/branch', 'Main'); // 'Main'
$reader->getAsInt('//seller/branchCode', 0);           // 0
```

Any Type

`get()` and `require()` read what the XPath finds as any `ValueType`. All the other methods are made with them:

```
use Derafu\Value\ValueType;

$reader->require('//info/number', ValueType::toInt());    // 21
$reader->get('//info/issued', ValueType::dateTime());   // DateTimeImmutable
```

What they find is what `XPathQuery::get()` gives: `null` if no node matches, a string for a node with only text, an array for a node with child elements, and a list if several nodes match.

Presence

```
$reader->hasKey('//seller/name');  // true: a node matches.
$reader->hasValue('//seller/name'); // true: a node matches and it has a value.
```

One Node or Many

XML has no list type: an element that appears once is a value, and an element that appears several times is a list. The reads of a list take what the XPath finds, one node or many, as a list:

```
$reader->getStringList('//seller/phone');  // ['111', '222']
$reader->getListOf('//line/qty', ValueType::toInt()); // [2, 3]
$reader->getList('//header');              // [['info' => [...], 'seller' => [...]]]:
one node, a list of one.
```

Method	Returns
`getList()`, `requireList()`	`list<mixed>`
`getStringList()`, `requireStringList()`	`list<string>`
`getListOf($xpath, $type)`, `requireListOf()`	`list<T>`, each item is of the type

A list of nothing is `null` with `getXxx()` and an exception with `requireXxx()`, as in `ArrayReader`. `$reader->getStringList('//seller/phone') ?? []` is the list that is empty when there is none.

Structures

A node with child elements is a structure, an array. An element without text and without children is an empty structure: it is `[]` in the methods that read a structure.

```
$reader->getMap('//seller');           // ['taxId' => 'TAX-1', 'name' => 'Seller
name', 'phone' => ['111', '222']]
$reader->getMap('//customs');           // []: an empty element.
$reader->getMap('//seller/name');    // throws: it has text, it is not a structure.
```

Method	Returns
`getArray()`, `requireArray()`	`array`
`getMap()`, `requireMap()`	`array<string, mixed>`
`getMapOf($xpath, $type)`, `requireMapOf()`	`array<string, T>`
`getArrayOf($xpath, $type)`, `requireArrayOf()`	`array<array-key, T>`

An empty element is also `[]` when it is an item of a list of structures or a value of a structure of structures, which is when the type of the items is `map`, `array` or `list`:

```
// <r><item/><item><a>1</a></item></r>
$reader->getListOf('//item', ValueType::map());  // [[], ['a' => '1']]

// <r><text/><text>x</text></r>
```

```
$reader->getListOf('//text', ValueType::string()); // ['', 'x']: the items are not
structures.
```

`get()` and `require()` give an empty element as it is in `XPathQuery`, an empty string.

Readers of a Node

`getReader()` gives a reader of one node, and `getReaders()` a reader of each node that the XPath matches, one node or many. They are readers like the one that makes them, scoped to the node: all that is read from them is read in the same way, at any depth.

```
foreach ($reader->getReaders('//line') ?? [] as $item) {
    $item->requireString('title');           // 'Item A', then 'Item B'.
    $item->getAsInt('qty');                 // 2, then 3.

    // Inside of a node, one element or many.
    foreach ($item->getReaders('code') ?? [] as $codigo) {
        $codigo->requireString('value');   // '111', '222', then '333'.
    }
}

$seller = $reader->requireReader('//seller');
$seller->requireString('taxId');
```

Method	Returns
`getReader()`, `requireReader()`	`XPathReader`; it throws if the XPath matches more than one node
`getReaders()`, `requireReaders()`	`list<XPathReader>`

The reader of an empty element is a reader of `[]`: everything that it reads is absent. A node that is not an element or that has text is not a structure and has no reader.

An optional `ValueType` checks the structure of the node: `$reader->getReader('//seller', ValueType::map())`.

Relative XPath

The XPath expressions of a reader of a node are relative to it: `title` and `./title` are its child. A `//` still starts at the root of the document:

```
$item->requireString('//seller/name'); // 'Seller name': from the root.
```

The Whole Structure

The operations on the whole structure of the document, or of the node of a reader, are the ones of `ArrayReader`:

```
$reader->toArray();                       // What XmlDocument::toArray() gives.
$reader->toMap();                         // The structure, that must have only string
keys.
$reader->to(ValueType::array());          // Checks it against a type.
$seller->assertOnlyKeys('taxId', 'name', 'phone'); // An error for another child.
```

Their errors are `XmlQueryException` like the ones of the reads, with the XPath of the structure: `/` for the whole document, and for an element that is not allowed the XPath of the element (`//seller/branch`).

Errors

Every error is an `XmlQueryException` (see Exceptions) that has the XPath expression, the document and, when it was a type that failed, the cause:

```
use Derafu\Xml\Exception\XmlQueryException;

try {
    $reader->requireString('//seller/branch');
} catch (XmlQueryException $e) {
    $e->getMessage();         // The XPath "//seller/branch" is required: it matched
no node.
    $e->getXpathExpression(); // '//seller/branch'
    $e->getXmlDocument();     // The document.
}
```

The errors of a reader of a node have the full XPath of the node: `//seller/missing`, or `(//line)[2]/title` for the second of several. With a source, the message says it: `The XPath "//seller/branch" of invoice.xml is required: it matched no node.`

An XPath expression that is not valid throws the `XmlQueryException` of `XPathQuery`.

Methods

Kind	Methods
Presence	`hasKey()`, `hasValue()`
Any type	`get()`, `require()`

Kind	Methods
Text and values	`getString()`, `getNonEmptyString()`, `getDateTime()`, `getEnum()` and their `require` pairs
Conversions	`getAsString()`, `getAsInt()`, `getAsFloat()`, `getAsNumber()`, `getAsBool()` and their `requireAs` pairs
Structures	`getArray()`, `getMap()`, `getMapOf()`, `getArrayOf()` and their `require` pairs
Lists	`getList()`, `getStringList()`, `getListOf()` and their `require` pairs
Readers of a node	`getReader()`, `getReaders()` and their `require` pairs
The whole structure	`toArray()`, `to()`, `toMap()`, `toList()`, `assertOnlyKeys()`

Last updated on 09/11/2026

XML Service

The XmlService class and its encoder, decoder and validator to convert and validate XML documents

Anonymous · 09/11/2026

XmlService Class

`XmlService` is the entry point of the library: it encodes, decodes and validates XML documents in one place. It gives the work to three components, that can also be used alone.

Service Structure

1. **XmlEncoder**: Converts PHP arrays to XML documents.
2. **XmlDecoder**: Converts XML documents to PHP arrays.
3. **XmlValidator**: Validates XML documents against XSD schemas.

`XmlServiceInterface` is the three interfaces together (`XmlEncoderInterface`, `XmlDecoderInterface` and `XmlValidatorInterface`).

Basic Usage

```
use Derafu\Xml\Service\XmlEncoder;
use Derafu\Xml\Service\XmlDecoder;
use Derafu\Xml\Service\XmlValidator;
use Derafu\Xml\Service\XmlService;

// Initialize the components.
$encoder = new XmlEncoder();
$decoder = new XmlDecoder();
$validator = new XmlValidator();

// Create the service.
$xmlService = new XmlService($encoder, $decoder, $validator);
```

In an application with the Symfony container, `resources/config/xml-services.yaml` of the library registers the interfaces with their classes.

Encoding (Array to XML)

The `encode()` method converts a PHP array to an XML document (see Array Structure):

```
$data = [
    'invoice' => [
        '@attributes' => [
            'id' => '12345',
            'date' => '2025-03-05'
        ],
        'client' => [
            'name' => 'Acme Corporation',
            'tax_id' => '123456789'
        ],
        'items' => [
            'item' => [
                [
                    '@attributes' => ['id' => '1'],
                    'description' => 'Product A',
                    'quantity' => '2',
                    'price' => '19.99'
                ],
                [
                    '@attributes' => ['id' => '2'],
                    'description' => 'Product B',
                    'quantity' => '1',
                    'price' => '29.99'
                ]
            ]
        ],
        'total' => '69.97'
    ]
];

$xmlDocument = $xmlService->encode($data);
```

`$xmlDocument->saveXml()` gives:

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice id="12345" date="2025-03-05">
  <client>
    <name>Acme Corporation</name>
    <tax_id>123456789</tax_id>
  </client>
  <items>
    <item id="1">
      <description>Product A</description>
      <quantity>2</quantity>
      <price>19.99</price>
    </item>
    <item id="2">
      <description>Product B</description>
      <quantity>1</quantity>
```

```
      <price>29.99</price>
    </item>
  </items>
  <total>69.97</total>
</invoice>
```

The encoding of the XML is UTF-8. To have another one: `$xmlDocument->setEncoding('ISO-8859-1')`.

The document that `encode()` gives is asked to format the XML that it saves, as in the example (an XML that is built from an array has no spaces between its elements). To save it in one line: `$xmlDocument->setFormatOutput(false)`.

An array that can not be an XML throws an `XmlEncoderException` (see Exceptions).

Using Namespaces

The namespaces are declared in the `@attributes` of the element, with `xmlns` for the default one and `xmlns:prefix` for the others. The names of the elements have the prefix:

```
$data = [
    'inv:invoice' => [
        '@attributes' => [
            'xmlns:inv' => 'http://example.com/invoice',
            'id' => '12345',
        ],
        'inv:client' => 'Acme Corporation',
    ],
];

$xmlDocument = $xmlService->encode($data);
```

This generates:

```
<?xml version="1.0" encoding="UTF-8"?>
<inv:invoice xmlns:inv="http://example.com/invoice" id="12345">
  <inv:client>Acme Corporation</inv:client>
</inv:invoice>
```

Decoding (XML to Array)

The `decode()` method converts an XML document to a PHP array:

```
// Assuming $xmlDocument is a Derafu\Xml\XmlDocument instance.
$array = $xmlService->decode($xmlDocument);

// Or starting from a DOMElement of the document.
```

```
$element = $xmlDocument->getDocumentElement();
$array = $xmlService->decode($element);
```

A document without document element is an empty array.

The array has what the XML has: attributes and the namespaces that the elements declare in `@attributes`, comments in `@comment`, and text exactly as it is. See Array Structure and Round Trip and Fidelity.

Handling Repeated Elements

Elements with the same name are collected into a list:

```
<root>
  <item>Value 1</item>
  <item>Value 2</item>
  <item>Value 3</item>
</root>
```

Becomes:

```
[
    'root' => [
        'item' => [
            'Value 1',
            'Value 2',
            'Value 3'
        ]
    ]
]
```

An element that appears once is its value, not a list of one.

Validation

The `validate()` method checks an XML document against an XSD schema:

```
use Derafu\Xml\Exception\XmlException;

try {
    $xmlService->validate($xmlDocument, '/path/to/schema.xsd');
    echo "XML is valid!";
} catch (XmlException $e) {
    echo "Validation failed: " . $e->getMessage();

    // The errors of libxml, as XmlError.
```

```
    foreach ($e->getErrors() as $error) {
        echo "- $error\n";
    }

    // The document that was validated.
    $document = $e->getXmlDocument();
}
```

The message has all the errors:

```
The XML validation failed using the schema schema.xsd: Line 1, column 0: Element
'wrong': No matching global declaration available for the validation root.
```

The errors are explained and translated from their code (see Exceptions), and the namespace of the names, that libxml writes between braces (`{urn:example}`), is taken out.

Schema Auto-Detection

If the XML document has an `xsi:schemaLocation` attribute, the schema path can be omitted: the schema is the path that comes after the namespace.

```
// XML with xsi:schemaLocation="http://example.com/ns /path/to/schema.xsd"
try {
    $xmlService->validate($xmlDocument); // Will use /path/to/schema.xsd
} catch (XmlException $e) {
    // Handle validation error.
}
```

The path has to be one of a file that exists. If the document does not have `xsi:schemaLocation`, or the path does not exist, an `XmlException` is thrown.

Integration Example

This example shows a typical workflow:

```
// Initialize the service.
$xmlService = new XmlService(new XmlEncoder(), new XmlDecoder(), new XmlValidator());

// Create XML from array.
$data = ['root' => ['element' => 'value']];
$xmlDocument = $xmlService->encode($data);

// Save to a file.
file_put_contents('document.xml', $xmlDocument->saveXml());

// Load from a file.
```

```
$loadedXml = new \Derafu\Xml\XmlDocument();
$loadedXml->loadXml(file_get_contents('document.xml'));

try {
    // Validate.
    $xmlService->validate($loadedXml, '/path/to/schema.xsd');

    // Convert back to array.
    $array = $xmlService->decode($loadedXml);

    // Process the data.
    // ...

} catch (\Derafu\Xml\Exception\XmlException $e) {
    // Handle validation errors.
}
```

Last updated on 09/11/2026

Exceptions

The exceptions of the XML library, when they are thrown, the details that they have and how to translate their messages

Anonymous · 09/11/2026

Exceptions

All the exceptions of the library extend `Derafu\Xml\Exception\XmlException`, so one `catch` takes all of them.

The Exceptions

Exception	When it is thrown
`XmlException`	A document without document element where it is needed, a validation that fails, a schema that is not found, and failures of PHP while the XML is generated (serializing, canonicalizing).
`XmlParseException`	`loadXml()` with an XML that is empty or is not well formed.
`XmlQueryException`	An XPath expression that is not valid or that fails, a read of `XPathReader` that finds no node or finds a value that is not of the type, and `C14NEncoded()` with an XPath that finds no node.
`XmlEncoderException`	An array that can not be converted to XML.

`XmlParseException`, `XmlQueryException` and `XmlEncoderException` extend `XmlException`.

Array that Can Not Be Converted

`XmlEncoder::encode()` throws `XmlEncoderException` when:

- The array has more than one element at its first level, or a list: an XML has only one root element.
- A name of an element or attribute is not text (a number is the position of a value in a list, not a name).
- A value is not text: a string, a number, a boolean or an object that can be a string are valid. An array can not be the value of an attribute.
- `@value` is at the first level of the array, outside of any element.
- A comment has two hyphens in a row or ends with a hyphen.

What the Exceptions Have

```
use Derafu\Xml\Exception\XmlException;

try {
    $document->loadXml('<r><a></r>');
} catch (XmlException $e) {
    $e->getMessage();     // "Failed to load the XML: Line 1, column 11: Opening and
ending tag mismatch: a (line 1) and r"
    $e->getErrors();      // A list of XmlError, one for each error of libxml.
    $e->getXmlDocument(); // The document that caused the exception, if there is one.
}
```

- `getErrors()` gives the errors that explain the exception, as a list of `XmlError`: the ones of libxml when an XML is loaded, canonicalized or validated. It is an empty list if there are none.
- `getXmlDocument()` gives the `XmlDocument` that caused the exception, or `null`.
- `XmlQueryException` also has `getXpathExpression()`: the XPath that was asked. In a read of `XPathReader` it is the full XPath of the node, and the exception also has the cause when it was a type that failed (`getPrevious()`).

When there are errors, the message has them, so the message says what failed. When an XML is loaded or canonicalized the message has the error that caused the failure: the first fatal one (the others that libxml reports after it are consequences of it), or else the first error. When an XML is validated the message has all the errors:

```
The XML validation failed using the schema schema.xsd: Line 1, column 0: Element
'wrong': No matching global declaration available for the validation root.
```

The Errors of libxml

An `XmlError` is an error that libxml found, with its data:

```
foreach ($e->getErrors() as $error) {
    $error->level;   // XmlErrorLevel::Warning, XmlErrorLevel::Error or
XmlErrorLevel::Fatal.
    $error->code;    // The code of the error in libxml, for example 76.
    $error->line;    // The line of the XML, 0 if it has none.
    $error->column;  // The column of the line, 0 if it has none.
    $error->message; // The original text of libxml.
    (string) $error; // "Line 1, column 11: Opening and ending tag mismatch: a (line
1) and r"
}
```

An `XmlError` is translatable like the exceptions: `$error->trans($translator)` says it in the language of the translator. The text is made from the code of the error: the library has the translation of each

error of libxml that it can say when it loads, canonicalizes or validates an XML (parsing, namespaces and XSD schemas). An error that is not known, or that libxml says in a way that is not known, keeps the original text of libxml. A warning says so (`Warning, line 1, column 4: ...`), and an error without a line (for example one of the schema) has no location.

The names of elements and attributes are shown without the namespace that libxml writes between braces (`{urn:example}`). The errors are in the exception for whoever wants to show them in another way: the code identifies the error better than its text.

Translating the Messages

The exceptions are translatable (they are the ones of the `derafu/translation` library): their messages are in English, and the library has their translation to Spanish. To have the message in another language, use `trans()` with a translator:

```
use Derafu\Translation\TranslatorFactory;
use Derafu\Xml\Translation\XmlTranslationResourceProvider;

$translator = TranslatorFactory::create('es', [], [new
XmlTranslationResourceProvider()]);

try {
    $document->loadXml('');
} catch (XmlException $e) {
    echo $e->getMessage();            // "The XML content that you want to load is
empty."
    echo $e->trans($translator);      // "El contenido XML que se quiere cargar está
vacío."
}
```

The translations are in `resources/translations/errors+intl-icu.es.php`. In an application with the Symfony container, `resources/config/xml-services.yaml` of the library registers `XmlTranslationResourceProvider` with the tag `derafu_translation.resource_provider`, so its translations are loaded with the ones of the other libraries.

The errors in `getErrors()` are translated too: see The Errors of libxml.