Docs

XML Project

Derafu XML

Anonymous · 24/08/2026 · #php

Table of Contents

XML Project

Derafu XML

Anonymous · 24/08/2026 · #php

Derafu XML

Last updated on 24/08/2026

Introduction

Library for XML manipulation

Anonymous · 24/08/2026

Library for XML manipulation

A comprehensive PHP library for XML manipulation, providing robust tools for encoding, decoding, validating, and querying XML documents.

Features

- **Conversion**: Transform between XML and PHP arrays in both directions.
- **Validation**: Validate XML documents against XSD schemas.
- **Querying**: Powerful XPath querying with parameter support.
- **Canonicalization**: Support for C14N and ISO-8859-1 encoding.
- **Encoding**: Proper handling of character encoding between UTF-8 and ISO-8859-1.
- **Special Characters**: Automatic handling of XML special characters and entities.

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 XML.
$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();

// Get a canonicalized version with ISO-8859-1 encoding.
$isoXml = $xmlDocument->C14NWithIso88591Encoding();
```

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 contains ['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']]
```

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(); // Get detailed error information.
}
```

Advanced Usage

Working with Namespaces

```
// Create XPath query with namespace support.
$namespaces = ['ns' => 'http://example.com/namespace'];
$query = new XPathQuery($xmlContent, $namespaces);

// Query with namespace.
$result = $query->get('//ns:Element');
```

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.

- Use `@value` for node text content when attributes are present.
- Arrays of values create repeated nodes with the same name.

Example:

```
$data = [
    'root' => [
        'element' => [
            '@attributes' => ['id' => '123'],
            '@value' => 'content',
        ],
        'items' => [
            'item' => ['value1', 'value2', 'value3'],
        ],
    ],
];
```

Produces:

```
<root>
  <element id="123">content</element>
  <items>
    <item>value1</item>
    <item>value2</item>
    <item>value3</item>
  </items>
</root>
```

Last updated on 24/08/2026

Array Structure

XML-Array Conversion Structure

Anonymous · 24/08/2026

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.

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 attributes are present.

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'
        ]
    ]
]
```

Special Cases

Empty Elements

Empty elements are represented as `null` or empty strings:

XML:

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

Array:

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

or when creating XML:

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

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.
4. The special key `@value` defines the text content when attributes are present.
5. Arrays of values create multiple elements with the same name.
6. Null, false, and empty arrays are skipped (not included in the XML).
7. Empty strings (’’) create empty elements.

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.
4. When attributes are present, text content is stored under the `@value` key.
5. Multiple elements with the same name are collected into arrays.
6. Empty elements become null values.
7. Complex nested structures are preserved in the array hierarchy.

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, check for null values to handle empty elements properly.

Last updated on 24/08/2026

XML Document

XmlDocument Class

Anonymous · 24/08/2026

XmlDocument Class

The `XmlDocument` class extends PHP's native `DOMDocument` with additional functionality for XML manipulation, transformation, and querying. This class is the core component for working with XML documents in the Derafu XML library.

Key Features

- Extended XML loading with character encoding handling.
- Simplified access to XML metadata (namespace, schema).
- Canonicalization support with encoding options.
- XPath query integration.
- Array conversion capabilities.
- Signature node handling for XML digital signatures.

Basic Usage

Creating a New Document

```
use Derafu\Xml\XmlDocument;

// Create with default version (1.0) and encoding (ISO-8859-1).
$document = new XmlDocument();

// Or specify version and encoding.
$document = new XmlDocument('1.0', 'UTF-8');
```

Loading XML Content

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

// The method handles encoding conversion automatically.
$utf8Xml = '<?xml version="1.0"
encoding="UTF-8"?><root><element>Árbol</element></root>';
```

```
$document->loadXml($utf8Xml); // Will convert to ISO-8859-1 if needed.
```

Accessing Document Information

```
// Get the root element name.
$rootName = $document->getName(); // e.g., "root"

// Get the XML namespace (if any).
$namespace = $document->getNamespace(); // e.g., "http://example.com"

// Get the schema location (if any).
$schema = $document->getSchema(); // e.g., "schema.xsd"
```

Saving and Serializing

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

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

// Get the canonicalized (C14N) version.
$canonXml = $document->C14N();

// Get canonicalized version with ISO-8859-1 encoding.
$isoCanonXml = $document->C14NWithIso88591Encoding();

// Get flattened canonicalized version (whitespace removed between tags).
$flatXml = $document->C14NWithIso88591EncodingFlattened();
```

XPath Querying

The XmlDocument class has integrated XPath querying capabilities:

```
// Execute an XPath query and get result as string or array.
$result = $document->query('/root/element');

// Get DOMNodeList from an XPath query.
$nodes = $document->getNodes('/root/element');

// Use parameters in XPath queries.
$params = ['id' => '123'];
$result = $document->query('/root/element[@id=:id]', $params);
```

Array Conversion

```
// 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');
```

Working with XML Digital Signatures

If the document contains an XML digital signature, you can extract it:

```
// Get the signature node as XML.
$signatureXml = $document->getSignatureNodeXml();

// Returns null if no signature is present.
if ($signatureXml !== null) {
    // Process signature...
}
```

Handling Special Characters and Entities

The `XmlDocument` class works with the `XmlHelper` to properly handle special characters and entities:

```
// When saving XML, entities are handled correctly.
$document->loadXml('<root><element>Text with & < > " \'</element></root>');
$xml = $document->saveXml();
// Produces: <root><element>Text with &amp; &lt; &gt; &quot; &apos;</element></root>
```

Advanced Canonicalization

Working with Specific Nodes

You can apply canonicalization to specific parts of the document:

```
// Canonicalize only a portion of the document.
$canonXml = $document->C14NWithIso88591Encoding('/root/section');
```

Differences Between Canonicalization Methods

The class offers several canonicalization methods:

1. **C14N()**: Standard canonicalization, always outputs UTF-8.
2. **C14NWithIso88591Encoding()**: Canonicalization with conversion to ISO-8859-1.
3. **C14NWithIso88591EncodingFlattened()**: ISO-8859-1 canonicalization with whitespace removal.

These methods are particularly useful when working with digital signatures or when XML needs to be processed by systems with specific encoding requirements.

Error Handling

The `loadXml()` method throws descriptive exceptions when it encounters errors:

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

try {
    $document->loadXml($potentiallyInvalidXml);
} catch (XmlException $e) {
    echo "Error loading XML: " . $e->getMessage();
    $errors = $e->getErrors(); // Get detailed error information if available.
}
```

Performance Considerations

- When working with large XML documents, prefer using XPath queries to extract specific sections rather than converting the entire document to an array.
- The canonicalization methods perform more processing, so use them only when needed.
- The `getXml()` method is more efficient than `saveXml()` when you only need the XML content without the declaration.

Last updated on 24/08/2026

XPath Query

XPathQuery Class

Anonymous · 24/08/2026

XPathQuery Class

The `XPathQuery` class is a powerful tool for extracting data from XML documents using XPath expressions. It provides a convenient interface to query XML with support for namespaces, parameterized queries, and complex data structures.

Basic Usage

```
use Derafu\Xml\XPathQuery;

// Initialize with XML string or 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');
```

Working with Namespaces

```
// Initialize with namespace support.
$namespaces = [
    'ns' => 'http://example.com/ns',
    'xs' => 'http://www.w3.org/2001/XMLSchema'
];

$query = new XPathQuery($xmlString, $namespaces);

// Use namespaces in queries.
$result = $query->get('//ns:Element/xs:Type');
```

Parameterized Queries

The `XPathQuery` class supports parameterized queries, making it safer and easier to build dynamic XPath expressions:

```
// Query with parameters.
$params = [
    'id' => '123',
    'type' => 'product'
];

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

This automatically handles escaping of values, including proper handling of values containing quotes.

Context Nodes

You can limit the scope of your query by providing a context node:

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

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

Return Value Handling

The `get()` method is intelligent about what it returns:

1. `null` if no matching nodes are found.
2. A string if a single node with no children is found.
3. An array if multiple nodes are found.
4. A structured array if nodes with child elements are found.

Example of Structured Results

For XML like:

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

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

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

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

Special Features

Using Without Namespaces

If you need to query XML with namespaces but don't want to specify them all, you can use the class without registering namespaces:

```
$query = new XPathQuery($xmlString); // No namespaces registered.

// This will match any element named 'Element' regardless of its namespace.
$result = $query->get('//Element');
```

Value Quoting

The class handles the quoting of values in XPath expressions automatically, even when values contain both single and double quotes:

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

Error Handling

The class provides clear error messages when there are issues with the XML document or XPath expressions:

```
try {
    $result = $query->get('//invalid[xpath');
} catch (InvalidArgumentException $e) {
    // Handle the error.
}
```

Last updated on 24/08/2026

XML Service

XmlService Class

Anonymous · 24/08/2026

XmlService Class

The `XmlService` class is the central component of the Derafu XML library, providing a unified interface for encoding, decoding, and validating XML documents. It follows a service-oriented architecture by delegating the actual work to specialized components.

Service Structure

The `XmlService` relies on three key components:

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

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);
```

Encoding (Array to XML)

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

```
$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);
```

This produces:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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>
```

Using Namespaces

You can specify XML namespaces during encoding:

```
$namespace = ['http://example.com/invoice', 'inv'];
$xmlDocument = $xmlService->encode($data, $namespace);
```

This generates:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<inv:invoice id="12345" date="2025-03-05" xmlns:inv="http://example.com/invoice">
  <!-- Content with namespace prefixes -->
</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.
$element = $xmlDocument->getDocumentElement();
$array = $xmlService->decode($element);
```

Handling Repeated Elements

By default, elements with the same name are collected into arrays:

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

Becomes:

```
[
    'root' => [
        'item' => [
            'Value 1',
```

```
            'Value 2',
            'Value 3'
        ]
    ]
]
```

You can control this behavior with the `$twinsAsArray` parameter:

```
$array = $xmlService->decode($xmlDocument, null, twinsAsArray: true);
```

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();

    // Get detailed error information.
    $errors = $e->getErrors();
    foreach ($errors as $error) {
        echo "- $error\n";
    }
}
```

Schema Auto-Detection

If the XML document includes a `schemaLocation` attribute, you can omit the schema path:

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

Error Translations

The validator can translate technical libxml error messages into more user-friendly messages:

```
$translations = [
```

```
    'element1' => 'Customer Info',
    'element2' => 'Product Details'
];

try {
    $xmlService->validate($xmlDocument, '/path/to/schema.xsd', $translations);
} catch (XmlException $e) {
    // Error messages will use the translated element names.
}
```

Integration Example

This complete example shows how to use the XML service for a typical workflow:

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

// 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'));

// Validate.
try {
    $xmlService->validate($loadedXml, '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 24/08/2026