Docs

Array Project

Typed array readers, composable validation types and array transformations

Anonymous · 15/11/2026 · #php

Table of Contents

Array Project 3
- ***Introduction*** 4
- ***Readers*** 7
- ***Writers*** 14
- ***Value Types*** 18
- ***Paths*** 20
- ***Casting*** 23
- ***Trees*** 27
- ***Tables*** 29
- ***Combinations*** 31
- ***Duplicates*** 32
- ***Translations*** 35

Array Project

Typed array readers, composable validation types and array transformations

Anonymous · 15/11/2026 · #php

Derafu Array

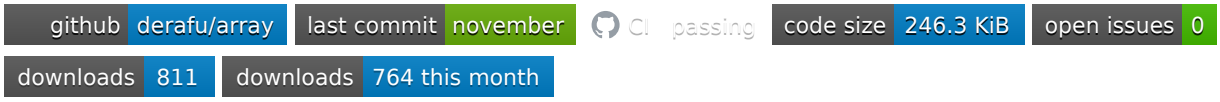
Last updated on 15/11/2026

Introduction

Typed array reading, composable validation types and explicit array transformations in PHP

Anonymous · 15/11/2026

Typed Array Reading and Transformations



`derafu/array` reads array values with explicit types, writes nested arrays in place, and provides transformations for paths, numeric text, trees, tables and subsets, and finds repeated items. It requires PHP 8.5 or later, `derafu/value` and `derafu/translation`.

Installation

```
composer require derafu/array
```

Reading Untrusted Data

```
use Derafu\Array\ArrayReader;

$reader = ArrayReader::fromValue(
    json_decode('{"host":"localhost","port":5432}', true),
    source: 'connection.json',
);

$host = $reader->requireString('host'); // string
$port = $reader->getInt('port', 5432);  // int
```

`fromValue()` rejects a value that is not an array. Use `fromArray()` when the input is already typed as an array. Readers are immutable snapshots of array contents; objects stored inside an array remain the same objects.

Reading is strict: `'5432'` is not an integer. Use getAsXXX()/requireAsXXX() for explicit field conversions, or ArrayCaster for its recursive array operation. Date and backed-enum types also interpret their supported representations.

Reading Nested Values

All typed getters and presence checks accept dot notation. An existing literal key matching the complete expression takes precedence over navigation.

```
$reader = ArrayReader::fromArray(['connection' => ['port' => 5432]]);
$reader->requireInt('connection.port'); // 5432
```

See Readers for presence and validation rules, and Paths for the same literal-key precedence when reading, setting and wrapping values.

Writing Values

```
use Derafu\Array\ArrayWriter;

$data = ['order' => ['total' => '1190']];
$writer = ArrayWriter::fromReference($data);

$writer->set('order.paid', true);
$writer->getReader()->requireAsInt('order.total'); // 1190
```

`ArrayWriter` writes in the array it receives by reference, with the same keys as the reader. See Writers.

The Classes

All eight array utility classes live in `Derafu\Array`. Composable value types and scalar conversions come from Derafu Value.

Class	Purpose	Guide
`ArrayReader`	Optional and required typed reads, nested readers and key presence.	Readers
`ArrayWriter`	In-place writes: set, remove, append, merge, defaults and scopes of an array.	Writers
`ArrayPath`	Nested access, dot notation, resolving a path into keys and wrapping a value in a list.	Paths
`ArrayCaster`	Recursive value interpretation delegated to ValueParser, and replacement of given or blank values by `null`.	Casting
`ArrayTree`	Parent-linked rows into trees and trees into outlines.	Trees
`ArrayTable`	Columns into rows, pairs into maps and keys into columns.	Tables

Class	Purpose	Guide
`ArrayCombinations`	Subsets of a list.	Combinations
`ArrayDuplicates`	Items of a list that repeat the values of another one.	Duplicates

`ArrayReader` is a `final readonly` object. `ArrayWriter` is a `final` object that writes in place in the array it holds by reference. The other array classes expose static transformations. Reader type arguments are instances of `Derafu\Value\ValueType`. Exceptions raised by the library's validation are translatable; see Translations.

Lists, Maps and Arrays

Type	Accepted keys
List	Consecutive integers starting at `0`.
Map	Strings only. Numeric strings that PHP converts to integer keys are rejected.
Array	Any PHP array keys: integers, strings or both, consecutive or not.

Empty arrays are accepted by all three. Choose `arrayOf()` when positions or identifiers must be preserved without requiring a list. Reading a nested array as a reader does not require it to be a map.

Last updated on 15/11/2026

Readers

Optional and required typed reads, null handling, integer keys, nested readers, sparse collections and typed views of the whole array

Anonymous · 15/11/2026

ArrayReader

```
use Derafu\Array\ArrayReader;
use Derafu\Value\ValueType;

$reader = ArrayReader::fromArray(['name' => 'Alice', 'enabled' => false]);
$reader->requireString('name'); // 'Alice'
$reader->getBool('enabled', true); // false
```

Creating a Reader

Factory	Input	Behavior
`fromArray(array $data, ?string $source = null)`	An array.	Creates a reader at the root.
`fromValue(mixed $data, ?string $source = null, string $path = '')`	Untrusted input.	Validates that it is an array, without converting it.
`new ArrayReader($data, $path = '', $source = null)`	An array.	Creates a reader with an explicit path and source.

`toArray()` returns the stored array. Changing the original array after creating a reader does not change the reader's array values. See The Whole Array to check its shape.

Optional and Required Values

A missing key or a value of `null` uses the optional getter's default. A present value of another type throws. Falsy values such as `false`, `0`, `''` and `[]` are preserved when they match the requested type.

Optional method	Required method	Value
`getString($key, $default = null)`	`requireString($key)`	String.

Optional method	Required method	Value
`getNonEmptyString($key)`	`requireNonEmptyString($key)`	String containing at least one non-whitespace character; returned without trimming.
`getInt($key, $default = null)`	`requireInt($key)`	Integer.
`getFloat($key, $default = null)`	`requireFloat($key)`	Float; stored integers are rejected.
`getNumber($key, $default = null)`	`requireNumber($key)`	Integer or float.
`getBool($key, $default = null)`	`requireBool($key)`	Boolean.
`getDateTime($key)`	`requireDateTime($key)`	`DateTimeInterface`; see Value Types.
`getObject($key, $class)`	`requireObject($key, $class)`	Instance of the class or interface; the return type is that class.
`getEnum($key, $class)`	`requireEnum($key, $class)`	Case of a backed enum, or its exact backing value; the return type is that enum. See Backed Enums.
`getArray($key)`	`requireArray($key)`	Any array.
`getIndexed($key)`	`requireIndexed($key)`	Array with integer keys, preserving gaps and order.
`getIndexedOf($key, $type)`	`requireIndexedOf($key, $type)`	Integer-keyed array whose values are parsed by the supplied type.
`getMap($key)`	`requireMap($key)`	Array with string keys.
`getList($key)`	`requireList($key)`	Zero-based consecutive list.
`getStringList($key)`	`requireStringList($key)`	List of strings.
`getIntList($key)`	`requireIntList($key)`	List of integers.
`getScalarList($key)`	`requireScalarList($key)`	List of booleans, numbers or strings.
`getArrayOf($key, $type)`	`requireArrayOf($key, $type)`	Array whose values match a type; keys are preserved.
`getMapOf($key, $type)`	`requireMapOf($key, $type)`	String-keyed array whose values match a type.
`getListOf($key, $type)`	`requireListOf($key, $type)`	List whose values match a type.

Optional methods return `null` when there is no value or default. The primitive getters with a non-null default have a non-null return type for PHPStan. Required methods reject missing keys and `null`, including when the requested `ValueType` is nullable.

The generic methods accept any composable type:

```
$reader = ArrayReader::fromArray(['mode' => 'fast']);
$reader->get('mode', ValueType::literal('fast', 'safe'), 'safe');
$reader->require('mode', ValueType::string());
$reader->get('tags', ValueType::listOf(ValueType::string()), []);
```

Defaults are returned as supplied, not validated or converted by the requested type.

Conversion variants exist only where a conversion exists: scalars (see below). Objects, arrays, dates and enums have no `getAsXXX()` variant: an object is never built from another value, and dates and enums already interpret their supported input forms.

Indexed Arrays

The indexed family delegates to `ValueType::indexed()` and `indexedOf()`. Integer keys may start anywhere, be negative, or have gaps. Lists and empty arrays are accepted. String keys are rejected without conversion. PHP itself normalizes some numeric string keys to integers when the input array is created.

```
$reader = ArrayReader::fromArray([
    'rows' => [8 => ['quantity' => '12.0'], 2 => ['quantity' => '1e3']],
], 'report.json');

$reader->requireIndexed('rows'); // The original rows and keys.
$reader->requireIndexedOf('rows', ValueType::mapOf(ValueType::toInt()));
// [8 => ['quantity' => 12], 2 => ['quantity' => 1000]]

$rows = $reader->requireIndexedReaders('rows', ValueType::map());
$rows[8]->requireAsInt('quantity'); // 12
```

Optional reads return `null` for missing or null values; required reads throw `MissingValueException`. Empty arrays return `[]`. Item types determine whether values are validated strictly or explicitly converted. Readers and the input array remain unchanged by these reads.

Errors preserve paths and sources, including a string key such as `rows.extra` or a value such as `rows.8.quantity`. The collection is not sorted or reindexed.

Explicit Conversions

Strict getters validate existing types. The getAsXXX()/requireAsXXX() methods explicitly convert through ValueType and ValueCaster from Derafu Value.

Optional method	Required method	Converted type
`getAsInt($key, $default = null)`	`requireAsInt($key)`	Integer.
`getAsFloat($key, $default = null)`	`requireAsFloat($key)`	Float.
`getAsNumber($key, $default = null)`	`requireAsNumber($key)`	Integer when possible, otherwise float.
`getAsBool($key, $default = null)`	`requireAsBool($key)`	Boolean.
`getAsString($key, $default = null)`	`requireAsString($key)`	String.

```
use Derafu\Array\ArrayReader;

$reader = ArrayReader::fromArray(['port' => '5432', 'enabled' => 'off', 'label' =>
false]);
$reader->requireAsInt('port');       // 5432
$reader->requireAsBool('enabled');   // false
$reader->requireAsString('label');   // '0'
$reader->getAsNumber('missing', 0);  // 0
```

Missing keys and null retain the reader's existing default/required rules. Present values that cannot be converted throw; they do not use the default. See ValueCaster's conversion contracts.

The Whole Array

The typed reads above take a key. To check the reader's own array, use `to()` with any composable type, or one of its shortcuts. They mirror `get()`, `getMap()`, `getIndexed()` and `getList()`, and they throw `TypeMismatchException` when the array is not of the type.

Method	Returns
`toArray()`	The stored array, unchecked: `array<array-key, mixed>`.
`toMap()`	The array when every key is a string: `array<string, mixed>`.
`toIndexed()`	The array when every key is an integer: `array<int, mixed>`.
`toList()`	The array when it is a consecutive list: `list<mixed>`.
`to($type)`	The array checked by any type, such as `ValueType::mapOf(ValueType::int())`.

```
$reader = ArrayReader::fromArray(['name' => 'Alice', 'age' => 30]);
```

```
$reader->toMap(); // ['name' => 'Alice', 'age' => 30]
$reader->to(ValueType::mapOf(ValueType::scalar()));

ArrayReader::fromArray(['a', 'b'])->toMap();
// Throws: The key "$" must be of type map, array given.
```

An empty array is accepted as a map, an indexed array and a list. Numeric string keys, which PHP stores as integers, are not map keys. The error says `$` for a reader at the root; for a nested reader it carries the reader's path and source, as in `The key "rows.0" of order.json must be of type map, array given.`

Presence and Allowed Keys

Every read accepts an integer or string key. An existing literal key takes precedence; otherwise, dot notation such as `'connection.port'` navigates nested arrays.

Method	Meaning
`hasKey($key)`	The key exists, even when its value is `null`.
`hasValue($key)`	The key exists and its value is not `null`.
`assertOnlyKeys(...$keys)`	Throws for any key not allowed; returns the same reader. It does not require the allowed keys to exist.

```
$reader = ArrayReader::fromArray([0 => null, 'enabled' => false]);
$reader->hasKey(0); // true
$reader->hasValue(0); // false
$reader->hasValue('enabled'); // true
$reader->assertOnlyKeys(0, 'enabled');
```

Integer allowed keys and their equivalent string forms are accepted by `assertOnlyKeys()`. This method checks actual keys at the current level; allowed names are literal and are not expanded into paths. To check nested keys, obtain a nested reader and call `assertOnlyKeys()` on it.

Nested Readers and Collections

`getReader()` and `requireReader()` read any nested array as another reader: maps, lists and sparse arrays all work. An optional second argument validates the selected array with a `ValueType` before creating its reader. Omitting it, or passing `null`, accepts any array as before.

```
$reader = ArrayReader::fromArray(['authors' => [['Alice', 'alice']]]);
$author = $reader->requireReader('authors')->requireReader(0);
$author->requireString(0); // 'Alice'
$author->getString(1); // 'alice'
```

Optional	Required	Outer collection
`getReaders($key)`	`requireReaders($key)`	A list of arrays, returned as a list of readers.
`getArrayReaders($key)`	`requireArrayReaders($key)`	An array of arrays, preserving arbitrary keys.
`getIndexedReaders($key)`	`requireIndexedReaders($key)`	An integer-keyed array of arrays, preserving gaps and row order.

Each record can itself be any array. The collection methods also accept an optional second `ValueType` argument: it validates each record without changing the outer collection's requirements. `getReaders()` and `requireReaders()` still require a consecutive list; `getArrayReaders()` and `requireArrayReaders()` preserve arbitrary keys; `getIndexedReaders()` and `requireIndexedReaders()` require integer keys without reindexing.

Optional methods return `null` for absent or null values without invoking the type. Required methods throw `MissingValueException` for those values before invoking it. Empty collections return `[]`.

```
use Derafu\Array\ArrayReader;
use Derafu\Value\ValueType;

$reader = ArrayReader::fromArray([
    'innerJoin' => ['table' => 'stock', 'condition' => 'products.id =
stock.product_id'],
    'questions' => [['text' => 'What is PHP?']],
]);

$join = $reader->requireReader('innerJoin', ValueType::map());
$table = $join->requireString('table'); // 'stock'.
$questions = $reader->requireReaders('questions', ValueType::map());
$text = $questions[0]->requireString('text'); // 'What is PHP?'.
```

The supplied type must produce an array. Examples include `ValueType::map()`, `ValueType::list()`, `ValueType::arrayOf()` and `ValueType::mapOf()`. Scalar-producing types are not valid arguments. A map requires string keys; a positional list is rejected when a map is requested.

Each selected array is parsed once during reader creation, and the parsed result becomes the child reader's data. Explicit conversion types can therefore prepare a reader's values without modifying the parent reader. Both validation errors and subsequent child reads retain the full path and source.

```
use Derafu\Array\ArrayReader;
use Derafu\Value\ValueType;

$reader = ArrayReader::fromArray([
    'translations' => [
        'es' => ['save' => 'Guardar'],
```

```
        'en' => ['save' => 'Save'],
    ],
]);

$languages = $reader->requireArrayReaders(
    'translations',
    ValueType::mapOf(ValueType::string()),
);
$label = $languages['es']->requireString('save'); // 'Guardar'.
```

```
$reader = ArrayReader::fromValue(
    ['rows' => [3 => ['name' => 'Alice'], 8 => ['name' => 123]]],
    source: 'account API',
    path: 'response',
);

$rows = $reader->requireArrayReaders('rows');
$rows[3]->requireString('name'); // 'Alice'
$rows[8]->requireString('name');
// Throws: The key "response.rows.8.name" of account API must be of type string, int
given.
```

`requireReaders('rows')` would reject this outer collection because its keys are not a consecutive list. No reader reindexes an arbitrary collection.

Last updated on 15/11/2026

Writers

Write nested arrays in place, create levels, remove keys, append to lists, merge values and defaults, and write in a scope of the array

Anonymous · 15/11/2026

ArrayWriter

`ArrayWriter` writes the keys of an array in place, as many times as needed. It is the counterpart of `ArrayReader`: it takes keys the same way and reports errors with the same messages and paths.

```
use Derafu\Array\ArrayWriter;

$data = ['order' => ['totals' => ['net' => 1000]]];
$writer = ArrayWriter::fromReference($data);

$writer->set('order.id', 10);
$writer->getWriter('order.totals')->set('total', 1190);

// $data is now:
// ['order' => ['totals' => ['net' => 1000, 'total' => 1190], 'id' => 10]]
```

Creating a Writer

Factory	Input	Behavior
`fromReference(array &$data, ?string $source = null)`	A variable holding an array.	Creates a writer at the root that writes in that variable.
`new ArrayWriter($data, $path = '', $source = null)`	A variable holding an array.	Creates a writer with an explicit path and source.

The writer does not copy the array: every write changes the variable it was created from. `source` tells whoever reads an error which array it is (a file, a request, a document), as in `ArrayReader::fromArray()`, and `path` says where the array is inside a bigger one.

`toArray()` returns the current array.

Writing

Method	Behavior
`set($key, $value)`	Sets a value, creating the levels that are missing.

Method	Behavior
`remove($key)`	Removes a key. If the path does not exist, nothing happens.
`append($key, $value)`	Adds a value at the end of the list at the key. The list is created if the key is absent or `null`.
`merge($values)`	Merges recursively: the values replace what is already there.
`mergeDefaults($defaults)`	Merges recursively: what is already there is kept and the defaults only fill what is missing.

All of them return the writer, so calls can be chained. Values are not typed when written: `null` is stored as `null`, and the type is what the reader asks for when the value is read.

```
$data = ['items' => ['a']];
$writer = ArrayWriter::fromReference($data);

$writer
    ->append('items', 'b')
    ->append('tags', 'new')
    ->set('meta.owner', 'Alice')
    ->remove('items.0');

// ['items' => [1 => 'b'], 'tags' => ['new'], 'meta' => ['owner' => 'Alice']]
```

Keys

Keys follow the rules of the reader and of `ArrayPath`: an existing literal key matching the complete string takes precedence; otherwise dots separate nested levels.

```
$data = ['profile.name' => 'Alice'];
$writer = ArrayWriter::fromReference($data);

$writer->set('profile.name', 'Bob'); // writes the literal key
// ['profile.name' => 'Bob']
```

Levels

`set()`, `append()` and `getWriter()` create the levels of the path that are absent or `null` as arrays. A level that exists and is not an array is an error: it is not replaced.

```
$data = ['order' => ['id' => 33]];
ArrayWriter::fromReference($data, 'document')->set('order.id.x', 1);
// TypeMismatchException:
// The key "order.id" of document must be of type array, int given.
```

`append()` also requires the value at the key to be a list. `remove()` never fails: a path that does not exist, or that goes through a value that is not an array, has nothing to remove.

Merging

`merge()` and `mergeDefaults()` use the semantics of `array_replace_recursive()`: nested arrays are merged by key, lists are merged by index and not concatenated, and `null` counts as a value.

```
$data = ['a' => ['x' => 1, 'y' => 2]];
$writer = ArrayWriter::fromReference($data);

$writer->merge(['a' => ['y' => 20, 'z' => 30]]);
// ['a' => ['x' => 1, 'y' => 20, 'z' => 30]]

$writer->mergeDefaults(['a' => ['x' => 0, 'w' => 4]]);
// ['a' => ['x' => 1, 'w' => 4, 'y' => 20, 'z' => 30]]
```

Unlike `set()`, they do not fail when a scalar meets an array: the value that is merged in replaces it in `merge()`, and the existing value is kept in `mergeDefaults()`.

Writing in a Scope

`getWriter($key)` returns a writer over the array at a key. What it writes is written in the array it comes from. The levels that are missing are created as arrays, so, unlike `ArrayReader::getReader()`, it never returns `null`.

```
$data = [];
$writer = ArrayWriter::fromReference($data);

$totals = $writer->getWriter('order.totals');
$totals->set('net', 1000);
$totals->mergeDefaults(['tax' => 0, 'exempt' => 0]);

// ['order' => ['totals' => ['net' => 1000, 'tax' => 0, 'exempt' => 0]]]
```

The writer of a scope knows its path from the root, so its errors and the reader it gives say the full path (`order.totals.net`).

The writer of a scope keeps a PHP reference to its level. While it is alive, copies of the array it writes in share that level.

Reading What Was Written

The writer does not read. `getReader()` returns an `ArrayReader` over the current data of the writer, with its path and source. It is a snapshot: ask for a new one after writing to read what was written.

```
$data = ['order' => ['total' => '1190']];
$writer = ArrayWriter::fromReference($data);

$writer->set('order.paid', true);
$writer->getReader()->requireAsInt('order.total'); // 1190
$writer->getReader()->requireBool('order.paid');   // true
```

Writer or ArrayPath

`ArrayPath::set()` is for a single write: it returns a new array and leaves its input unchanged. Use `ArrayWriter` when an array is written several times, or when the writes happen at different levels of the same array.

Last updated on 15/11/2026

Value Types

Compose Derafu Value types for optional and required array reads

Anonymous · 15/11/2026

Value Types in Array Readers

ArrayReader accepts `Derafu\Value\ValueType<T>` for generic reads and collections. The type system belongs to Derafu Value, including primitive checks, explicit conversions, dates, backed enums, unions, nullable types and custom types.

```
use Derafu\Array\ArrayReader;
use Derafu\Value\ValueType;

$reader = ArrayReader::fromArray([
    'quantities' => [3 => '12.0', 8 => '2'],
    'mode' => 'fast',
], 'settings.php');

$reader->requireIndexedOf('quantities', ValueType::toInt());
// [3 => 12, 8 => 2]

$reader->get('mode', ValueType::literal('fast', 'safe'), 'safe');
// 'fast'
```

Validation and Conversion

`ValueType::int()` checks an existing integer; `ValueType::toInt()` requests conversion through ValueCaster. Built-in getters delegate to these same types: requireInt() is strict, while requireAsInt() converts explicitly. PHPStan follows the generic return type through these operations.

Arrays preserve arbitrary integer/string keys. Indexed arrays require integer keys and preserve gaps and order. Maps require string keys; lists require consecutive integer keys starting at zero. Empty collections are accepted. A type such as mapOf() checks every value, not a schema of required field names.

Nullability and Presence

A nullable type accepts null when parsed directly, but reader methods apply their own presence rules first: optional reads return the default for a missing or null key, and required reads reject both. `hasKey()` distinguishes a present null key from a missing key.

```
use Derafu\Array\ArrayReader;
use Derafu\Value\ValueType;

$reader = ArrayReader::fromArray(['count' => null]);
$reader->get('count', ValueType::toInt()->nullable(), 0); // 0
$reader->hasKey('count');   // true
$reader->hasValue('count'); // false
```

Backed Enums

```
use Derafu\Array\ArrayReader;
use Derafu\Value\ValueType;

enum Status: string
{
    case Pending = 'pending';
    case Approved = 'approved';
}

$reader = ArrayReader::fromArray(['status' => 'approved']);
$reader->require('status', ValueType::backedEnum(Status::class));
// Status::Approved
```

`getEnum()` and `requireEnum()` are shortcuts for the same type and return the enum class given:

```
$reader->requireEnum('status', Status::class); // Status::Approved
$reader->getEnum('missing', Status::class);    // null
```

Backing values use their exact int/string type, without scalar coercion. See ValueType for enum factories, dates, custom type callbacks and union order.

Error Context

ValueType reports the full nested path and the expected element type. ArrayReader adapts a value type mismatch to `Derafu\Array\Exception\TypeMismatchException`, preserving the Value exception as its cause. The reader keeps its own translated key-oriented messages and source context. See Translations.

Last updated on 15/11/2026

Paths

Read and update nested arrays, transform dot notation and wrap values in lists

Anonymous · 15/11/2026

ArrayPath

`ArrayPath` provides static operations over nested arrays. The transformations return arrays without modifying their input. To write several times in the same array, in place, see Writers.

```
use Derafu\Array\ArrayPath;

$data = ['customer' => ['name' => 'Alice', 'note' => null]];
ArrayPath::get($data, 'customer.name'); // 'Alice'
ArrayPath::get($data, ['customer', 'note'], 'default'); // null
ArrayPath::get($data, 'customer.missing', 'default'); // 'default'

$updated = ArrayPath::set($data, 'customer.address.city', 'Santiago');
// The original $data is unchanged.
```

Reading and Setting Paths

`get($array, $path, $default = null)`, `set($array, $path, $value)` and `wrap($array, $path)` accept a dot-separated string or a non-empty list of string segments.

`get()` returns a stored `null` unchanged. Its default is used when a key is absent or an intermediate level is not an array. `set()` creates missing levels and replaces non-array intermediate values with arrays.

For a string path, all three methods first use an existing literal key matching the complete string. Only when that key is absent do dots separate nested levels. A literal key with a null value still takes precedence.

```
$data = [
    'profile.name' => 'Alice',
    'profile' => ['name' => 'Nested'],
];

ArrayPath::get($data, 'profile.name'); // 'Alice'

$updated = ArrayPath::set($data, 'profile.name', 'Bob');
// ['profile.name' => 'Bob', 'profile' => ['name' => 'Nested']]
```

```
ArrayPath::get($updated, 'profile.name'); // 'Bob'
```

When the literal key is absent, `set()` creates or updates the nested path. Its interpretation therefore depends on the existing keys, including when starting from an empty array:

```
ArrayPath::set([], 'profile.name', 'Bob');
// ['profile' => ['name' => 'Bob']]
```

Literal matching applies to the complete string at the supplied array's root. After falling back to navigation, dots separate individual segments. Segment lists describe an explicit sequence of keys, including intermediate keys containing dots:

```
ArrayPath::get(['a.b' => ['c' => 1]], ['a.b', 'c']); // 1
ArrayPath::get(['items' => [['id' => 7]]], 'items.0.id'); // 7
```

`get()` returns `mixed`; use `ArrayReader` or `ValueType` when values must be validated.

Resolving a Path

`keys($array, $path)` returns the keys that a path names in a given array, with the same rule that `get()`, `set()`, `wrap()`, `ArrayReader` and `ArrayWriter` follow: an existing literal key matching the complete string is a single key; otherwise the string is split by its dots. A list of segments is returned as it is.

```
ArrayPath::keys(['a.b' => 1], 'a.b'); // ['a.b']
ArrayPath::keys([], 'a.b.c');          // ['a', 'b', 'c']
ArrayPath::keys([], ['a.b', 'c']);     // ['a.b', 'c']
```

Flat and Nested Forms

```
ArrayPath::flatten(['a' => ['b' => 1, 'c' => [2, 3]]]);
// ['a.b' => 1, 'a.c.0' => 2, 'a.c.1' => 3]

ArrayPath::flatten(['b' => 1], 'a');
// ['a.b' => 1]

ArrayPath::expand(['a.b' => 1, 'a.c' => 2, 'd' => 3]);
// ['a' => ['b' => 1, 'c' => 2], 'd' => 3]
```

`expand()` keeps integer keys at the current level and replaces conflicting scalar levels when a later path needs an array. `flatten()` emits leaf values; empty nested arrays produce no entry. Dots in literal keys are not escaped. These transformations are not a lossless round trip for every possible array.

Wrapping a Value

`wrap()` places a value in a one-element list unless it is already a list. It uses `array_is_list()`: having a key `0` alone does not make an array a list.

```
ArrayPath::wrap(['items' => ['name' => 'Alice']], 'items');
// ['items' => [['name' => 'Alice']]]

ArrayPath::wrap(['items' => [0 => 'a', 2 => 'b']], 'items');
// ['items' => [[0 => 'a', 2 => 'b']]]

ArrayPath::wrap(['document.reference' => ['id' => 123]], 'document.reference');
// ['document.reference' => [['id' => 123]]]
```

As with `get()` and `set()`, an existing complete literal key takes precedence over the nested path. An empty array and an existing list are unchanged. A missing path, a null value at the target, or a non-array intermediate level is left unchanged. Assign the returned array when applying a transformation.

Last updated on 15/11/2026

Casting

Recursively interpret array values through ValueParser, or replace given or blank values by null

Anonymous · 15/11/2026

ArrayCaster

`ArrayCaster` traverses array leaves and delegates each value directly to `Derafu\Value\ValueParser::parse()`. Its options and defaults match the parser:

```
ArrayCaster::cast(
    array $array,
    bool $preserveNumericFormat = false,
    bool $preserveNumericPrecision = false,
): array
```

The returned array keeps its original keys and nesting. The supplied array is not modified. Empty nested arrays remain empty arrays. Scalar interpretation, Stringable conversion and unsupported values follow ValueParser.

Recursive Interpretation

```
use Derafu\Array\ArrayCaster;

$input = [
    'quantity' => '12.0',
    'message' => ' text ',
    'blank' => '   ',
    'empty' => 'empty',
    'nothing' => '(null)',
    'nested' => [3 => ['enabled' => 'off', 'zero' => '0']],
];

$result = ArrayCaster::cast($input);
// [
//     'quantity' => 12,
//     'message' => ' text ',
//     'blank' => false,
//     'empty' => '',
//     'nothing' => null,
//     'nested' => [3 => ['enabled' => false, 'zero' => 0]],
// ]
```

There is no trimming or empty-value substitution before the parser. Unrecognized text keeps its spaces, while whitespace-only text becomes `false` through ValueCaster's boolean conversion. Integers are preferred when exact conversion is possible; numeric `'0'` and `'1'` therefore become integers rather than booleans.

Native numbers, booleans, nulls, resources and objects without Stringable remain unchanged. Stringable leaves are converted to text and interpreted using the same options. A failing Stringable conversion propagates Value's translatable `ValueConversionException` with its original cause.

Preserving Numeric Format

Pass `preserveNumericFormat: true` to keep numeric text when converting the resulting number back to string would alter its original spelling:

```
use Derafu\Array\ArrayCaster;

ArrayCaster::cast(
    ['a' => '12', 'b' => '19.90', 'c' => '007', 'd' => '1e3', 'e' => ' 12 '],
    preserveNumericFormat: true,
);
// ['a' => 12, 'b' => '19.90', 'c' => '007', 'd' => '1e3', 'e' => ' 12 ']
```

Exterior whitespace and decimal trailing zeros are part of the original spelling. This option concerns numeric text; special literals and boolean words are still interpreted normally.

Preserving Numeric Precision

Pass `preserveNumericPrecision: true` to retain numeric text when the parser detects lost significant decimal digits or a non-finite result:

```
use Derafu\Array\ArrayCaster;

ArrayCaster::cast(['large' => '1e400']);
// ['large' => INF]

ArrayCaster::cast(
    ['large' => '1e400', 'small' => '1e-400', 'fraction' => '1.0000000000000000001'],
    preserveNumericPrecision: true,
);
// ['large' => '1e400', 'small' => '1e-400', 'fraction' => '1.0000000000000000001']
```

Both options may be enabled together. Their exact guarantees, including the distinction between significant decimal digits and exact binary representation, are defined by ValueParser. ArrayCaster implements no separate numeric grammar or preservation rules.

Replacing Values by Null

`ArrayCaster::nullify()` traverses array leaves and replaces by `null` those that are one of the given values:

```
ArrayCaster::nullify(
    array $array,
    array $values,
): array
```

The list of values must have at least one value. The comparison is strict: `0` is not `false` and `'0'` is not `''`. The returned array keeps its original keys and nesting, empty nested arrays remain empty arrays and the supplied array is not modified. The caller decides which values mean “no value”; the method knows no rule of its own.

```
use Derafu\Array\ArrayCaster;

$input = [
    'net' => 100000,
    'vat' => false,
    'name' => '',
    'zero' => 0,
    'items' => [['code' => 1, 'amount' => false]],
];

ArrayCaster::nullify($input, [false, '']);
// [
//     'net' => 100000,
//     'vat' => null,
//     'name' => null,
//     'zero' => 0,
//     'items' => [['code' => 1, 'amount' => null]],
// ]
```

It prepares data for ArrayReader, where only `null` or a missing key is absent:

```
$reader = ArrayReader::fromArray(ArrayCaster::nullify($input, [false, '']));

$reader->getAsInt('vat', 0); // 0
```

Replacing Blank Values by Null

`ArrayCaster::nullifyBlank()` replaces by `null` the values that ValueInspector::isBlank() considers blank: an empty string or an empty array.

```
ArrayCaster::nullifyBlank(
    array $array,
): array
```

False, zero and whitespace-only strings are not blank, so they are kept. The arrays are traversed, but an array is blank only when it is empty: one that has only blank values is kept, with those values replaced by `null`. The returned array keeps its original keys and the supplied array is not modified.

```
use Derafu\Array\ArrayCaster;

ArrayCaster::nullifyBlank([
    'name' => '',
    'tags' => [],
    'active' => false,
    'quantity' => 0,
    'nested' => ['code' => '', 'items' => ['']],
]);
// [
//     'name' => null,
//     'tags' => null,
//     'active' => false,
//     'quantity' => 0,
//     'nested' => ['code' => null, 'items' => [null]],
// ]
```

Use `nullify()` to replace other values, such as `false` or a text that means “no value” in the source.

Reading and Conversion

Use ArrayReader’s strict getters to validate existing PHP types, or getAsXXX()/requireAsXXX() to convert individual fields. Use ArrayCaster when an entire array should be traversed. Recursive interpretation does not declare a required field schema or choose application defaults.

Last updated on 15/11/2026

Trees

Build parent-linked array trees and extract keyed outlines with nesting levels

Anonymous · 15/11/2026

ArrayTree

Rows into a Tree

`ArrayTree::fromRows($items, $parentField, $childrenField, $parentId = null)` uses the array keys as node identifiers. Each row's parent field refers to the key of its parent.

```
use Derafu\Array\ArrayTree;

$items = [
    1 => ['parent_id' => null, 'name' => 'root'],
    2 => ['parent_id' => 1, 'name' => 'child'],
    3 => ['parent_id' => 2, 'name' => 'leaf'],
];

$tree = ArrayTree::fromRows($items, 'parent_id', 'children');
// [1 => ['name' => 'root', 'children' => [
//     2 => ['name' => 'child', 'children' => [
//         3 => ['name' => 'leaf', 'children' => []],
//     ]],
// ]]]
```

Keys are preserved, parent fields are removed, and the children field is present even when empty. Parent comparisons are strict: integer `1` and string `'1'` are different parent values.

Rows without the parent field, and branches not reachable from the selected root parent, are omitted. Supply rows keyed by their identifiers; for database lists, `array_column($rows, null, 'id')` can prepare that structure.

Tree into an Outline

`toOutline($tree, $nameField, $childrenField)` traverses depth first and returns each node's name and nesting level, keeping its key:

```
ArrayTree::toOutline($tree, 'name', 'children');
// [
```

```
//     1 => ['name' => 'root', 'level' => 0],
//     2 => ['name' => 'child', 'level' => 1],
//     3 => ['name' => 'leaf', 'level' => 2],
// ]
```

This is an outline, not a reconstruction of the original rows: other fields are not included. Node keys must be unique across the tree to avoid later nodes replacing earlier outline entries.

A non-array node or a node with a missing or null name is skipped with its descendants. A node whose children field is not an array is included, but those children are omitted. The optional `$level` and by-reference `$result` parameters support recursive accumulation; ordinary callers use the first three arguments.

Last updated on 15/11/2026

Tables

Transform column groups into rows, key-value pairs into arrays and row keys into columns

Anonymous · 15/11/2026

ArrayTable

```
use Derafu\Array\ArrayTable;
```

Columns into Rows

`columnsToRows($array, $keys = null)` transposes column groups into rows. Columns are read at zero-based positions.

```
ArrayTable::columnsToRows(['name' => ['Alice', 'Bob'], 'age' => [30]]);
// [
//     ['name' => 'Alice', 'age' => 30],
//     ['name' => 'Bob', 'age' => null],
// ]

ArrayTable::columnsToRows(['name' => ['Alice']], ['name', 'email']);
// [['name' => 'Alice', 'email' => null]]
```

The longest selected column determines the number of rows. Missing positions and selected columns that do not exist supply `null`. If no selected column exists, the result is empty. With no `$keys`, every column is selected.

Pairs into a Map

`pairsToMap($table)` requires each row to be an array with exactly two columns. The first column becomes the key and the second its value. Column order determines their role, including for rows with named columns.

```
ArrayTable::pairsToMap([['name', 'Alice'], ['age', 30]]);
// ['name' => 'Alice', 'age' => 30]
```

Keys may be scalar or null. Integers stay integers; other accepted keys are converted to strings before PHP applies its array-key rules. For example, `null` becomes `''`, `true` becomes `'1'` (stored as integer `1`) and `2.7` becomes `'2.7'`.

Later duplicate keys replace earlier values. Invalid rows, or keys that are arrays, objects or resources, throw a translatable `Derafu\Array\Exception\ArrayException`, which extends `InvalidArgumentException`. Despite its name, this transformation can produce integer keys; use `ValueType::map()` separately if string-only keys are required.

Row Keys into a Column

`withKeyColumn($data, $idField)` adds each row's array key as a column:

```
ArrayTable::withKeyColumn(['u1' => ['name' => 'Alice'], 'u2' => ['name' => 'Bob']],
'id');
// ['u1' => ['id' => 'u1', 'name' => 'Alice'], 'u2' => ['id' => 'u2', 'name' =>
'Bob']]
```

Outer keys are preserved. A row that already has the named column keeps its own value: the operation supplies a default identifier rather than overwriting it. All three transformations return new arrays.

Last updated on 15/11/2026

Combinations

Generate subsets of a list with a minimum number of elements

Anonymous · 15/11/2026

ArrayCombinations

`ArrayCombinations::subsets($array, $minLength = 1)` enumerates subsets of a zero-based consecutive list. Each result is itself a list.

```
use Derafu\Array\ArrayCombinations;

ArrayCombinations::subsets([1, 2, 3]);
// [[1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

ArrayCombinations::subsets([1, 2, 3], 2);
// [[1, 2], [1, 3], [2, 3], [1, 2, 3]]

ArrayCombinations::subsets([1, 2], 0);
// [[], [1], [2], [1, 2]]
```

Selection is by position, preserving the input order within each subset. Equal values at different positions can therefore produce equal-looking subsets. The input is unchanged.

The algorithm examines `2^n` combinations. By default it excludes the empty subset, returning `2^n - 1` results; a minimum length of zero includes it. Use small lists: raising the minimum length filters results but does not avoid examining the combinations.

Last updated on 15/11/2026

Duplicates

Find the items of a list that repeat the values of another one, comparing by paths at any level

Anonymous · 15/11/2026

ArrayDuplicates

`ArrayDuplicates` finds the items of a list that repeat the values of an earlier item. What identifies an item are the values at the paths that are given, which can be at any level; without paths, the whole item is compared.

```
use Derafu\Array\ArrayDuplicates;

$documents = [
    ['TpoDoc' => 33, 'NroDoc' => 21],
    ['TpoDoc' => 33, 'NroDoc' => 22],
    ['TpoDoc' => 61, 'NroDoc' => 21],
    ['TpoDoc' => 33, 'NroDoc' => 21],
];

ArrayDuplicates::find($documents, ['TpoDoc', 'NroDoc']);
// [['index' => 3, 'first' => 0, 'value' => [33, 21]]]
```

Method	What it does
`find($items, $paths = [], $normalize = null)`	Every item that repeats an earlier one.
`first(...)`	The first one, or `null` if there is none.
`assertUnique(...)`	Throws `DuplicateValueException` for the first one. Returns nothing.

What Is Reported

Each repeated item is reported with three entries:

Entry	Content
`index`	The key of the repeated item in the list, as it is in the list.
`first`	The key of the first item that has those values.
`value`	The values read from the repeated item, in the order of the paths, as they are (before normalizing them).

The items are reported in the order of the list. An item that appears three times is reported twice, each one pointing to the first.

Paths

A path is a string, an integer or a list of them, and it is read with `ArrayPath::get()`: a key, dot notation (`'Encabezado.IdDoc.Folio'`) or the levels as a list (`['Encabezado', 'IdDoc', 'Folio']`). Paths of different levels can be mixed in the same call.

```
ArrayDuplicates::find($items, ['a', 'b.c']);
```

A path that is not in an item gives `null`, so two items that do not have it have the same value for it. An item that is an object is read through its public properties. An item that is neither an array nor an object can not be read by paths and is an `ArrayException`, as is a path that is not valid.

Without paths the whole item is compared, whatever it is: an array, an object or a scalar.

Comparison

Two items are the same when their values have the same type and the same value: `'1'` and `1` are different. The values must be serializable; one that is not (a closure, for instance) is an `ArrayException` that tells which item has it.

To compare in another way, pass `$normalize`, a `callable(mixed): mixed` that is applied to each value that is read before comparing it:

```
$normalize = static fn (mixed $value): string => (string) (int) trim((string) $value);

ArrayDuplicates::find([['n' => 21], ['n' => '0021'], ['n' => ' 21 ']], ['n'],
$normalize);
// [
//     ['index' => 1, 'first' => 0, 'value' => ['0021']],
//     ['index' => 2, 'first' => 0, 'value' => [' 21 ']],
// ]
```

The items are compared in one pass, so the time grows with the size of the list.

Requiring Unique Items

`assertUnique()` is for who only needs to know that there are no repeated items. Like `assertOnlyKeys()`, it checks a property of the whole data and throws a specific exception:

```
use Derafu\Array\Exception\DuplicateValueException;
```

```
try {
    ArrayDuplicates::assertUnique($documents, ['TpoDoc', 'NroDoc']);
} catch (DuplicateValueException $exception) {
    echo $exception->getMessage();
    // The item with key 3 repeats the item with key 0 (TpoDoc = 33, NroDoc = 21).

    $exception->getIndex();  // 3
    $exception->getFirst();  // 0
    $exception->getValues(); // [33, 21]
}
```

`DuplicateValueException` extends `ArrayException` and carries the key of the repeated item, the key of the first one and the values, so whoever catches it can say it in its own words. The message names the items by their key in the list, which can be a number or a text, and shows each path with the value that is repeated. Without paths it is `The item with key {index} is the same as the item with key {first}.` See Translations for its messages.

Last updated on 15/11/2026

Translations

Translate array validation errors and register the package resource provider

Anonymous · 15/11/2026

Translations

Errors from ArrayReader, ArrayTable and ArrayDuplicates belong to `Derafu\Array\Exception\ArrayException`, which extends `Derafu\Translation\Exception\Logic\TranslatableInvalidArgumentException`. MissingValueException, TypeMismatchException and UnexpectedKeyException specialize array read failures, and DuplicateValueException the items that repeat the values of another one. They can be caught as PHP's `InvalidArgumentException` and translated through Derafu Translation.

```
use Derafu\Array\ArrayReader;

try {
    ArrayReader::fromArray(['port' => 'abc'], 'settings.php')->requireInt('port');
} catch (InvalidArgumentException $exception) {
    echo $exception->getMessage();
    // The key "port" of settings.php must be of type int, string given.
}
```

ArrayCaster delegates to ValueParser and propagates its ValueConversionException when Stringable conversion fails. Register Value's provider when translating those exceptions; see Value Translations.

Messages and Context

Message	Cause
`The key "{key}" must be of type {expected}, {given} given.`	A value has an invalid type.
`The key "{key}" is required.`	A required read finds a missing or null value.
`The key "{key}" is not allowed; the allowed keys are: {allowed}.`	`assertOnlyKeys()` finds an unexpected key.
`The item with key {index} repeats the item with key {first} ({values}).`	`assertUnique()` finds a repeated item that was compared by paths; `{values}` has each path with its value (`TpoDoc = 33, NroDoc = 21`).
`The item with key {index} is the same as the item with key {first}.`	`assertUnique()` finds a repeated item that was compared whole.

Message	Cause
`The path {position} must be a string, an integer or a list of them, {given} given.`	A path of ArrayDuplicates is not valid.
`The item with key {index} must be an array or an object to read {paths}, {given} given.`	ArrayDuplicates reads paths from an item that is neither.
`The values of the item with key {index} can not be compared: {reason}.`	A value of an item can not be serialized to be compared.
`Row {index} must have exactly 2 columns.`	`pairsToMap()` finds an invalid row.
`The first column of row {index} must be a scalar or null to be a key.`	A pair has an invalid key value.

Reader and type messages also have variants containing `of {source}`. Nested readers retain the source and append their keys to the diagnostic path. Expected type names come from ValueType; `map` refers to string-keyed arrays. ArrayReader wraps Value type mismatches while preserving their nested path, element type and original cause. Direct ValueType calls use the Value exception contract.

Resource Provider

The package ships Spanish translations in `resources/translations/errors+intl-icu.es.php`, with the provider `Derafu\Array\Translation\ArrayTranslationResourceProvider`.

```
use Derafu\Array\Translation\ArrayTranslationResourceProvider;
use Derafu\Translation\TranslatorFactory;

$translator = TranslatorFactory::create(
    defaultLocale: 'es',
    fallbackLocales: ['es', 'en'],
    resourceProviders: [new ArrayTranslationResourceProvider()],
);
```

A translatable exception can then be translated with `$exception->trans($translator)`. The resource provider supplies translations; the library does not create a translator automatically.

Dependency Injection

Applications using Symfony DependencyInjection can import:

```
imports:
    - { resource: '../vendor/derafu/array/resources/config/array-services.yaml' }
```

The recipe tags `ArrayTranslationResourceProvider` with `derafu_translation.resource_provider`.

Pass providers with that tag to `TranslatorFactory::create()` through its `$resourceProviders` argument, following the setup in Derafu Translation.

Last updated on 15/11/2026