Docs

Container Project

Derafu Container

Anonymous · 15/11/2026 · #php

Table of Contents

Container Project 3
- ***Introduction*** 4
- ***Containers*** 7
- ***Typed Readers*** 11
- ***Validation*** 15
- ***Translations*** 20

Container Project

Derafu Container

Anonymous · 15/11/2026 · #php

Derafu Container

Last updated on 15/11/2026

Introduction

Specialized PHP data containers, from simple bags to stores validated with JSON Schema, with a common interface and typed array readers

Anonymous · 15/11/2026

Flexible Data Containers for PHP



A collection of specialized PHP data containers that provide different levels of structure and validation, from simple bags to schema-validated stores.

Features

- Purpose-built containers for different needs.
- Validation with a simple schema (`Vault`) or with JSON Schema (`Store`).
- Validation on every change: a container never holds data that breaks its schema.
- Typed readers through `getReader()`, with validation and explicit conversions understood by PHPStan.
- Simple to complex data structures, with dot notation.
- Sequential data handling.
- Common interface across containers.
- Translatable exceptions.
- Minimal dependencies.
- Comprehensive test coverage.

Why Derafu\Container?

Unlike generic data structures, Derafu\Container provides specialized containers each designed for specific use cases:

- **Bag**: Simple, flexible data storage without restrictions.
- **Vault**: Structured data validated with a simple schema (types, required keys, choices, defaults, normalizers).
- **Store**: Structured data validated with JSON Schema.
- **Journal**: Sequential data storage with LIFO access.

All of them share the same interface, so code that reads data does not care which container it

received.

Installation

Install via Composer:

```
composer require derafu/container
```

A Quick Tour

```
use Derafu\Container\Vault;

$options = new Vault(
    ['host' => 'imap.example.com'],
    [
        'host' => ['types' => 'string', 'required' => true],
        'port' => ['types' => 'int', 'default' => 993],
    ]
);

$host = $options->getReader()->requireString('host'); // string: "imap.example.com"
$port = $options->getReader()->requireInt('port');    // int: 993 (the default of the
schema).

try {
    $options->set('port', 'abc'); // Rejected: the port must be an int.
} catch (\Symfony\Component\OptionsResolver\Exception\InvalidOptionsException
$exception) {
    $options->getReader()->requireInt('port'); // Still 993: the failed change did
nothing.
}
```

Container Comparison

Feature	Bag	Vault	Store	Journal
Schema	No	Simple	JSON	No
Validation on every change	No	Yes	Yes	No
Defaults and normalizers	No	Yes	Defaults	No
Nested data	Yes	Yes	Yes	No
Dot notation	Yes	Yes	Yes	No
Sequential	No	No	No	Yes

Feature	Bag	Vault	Store	Journal
Typed reader	Yes	Yes	Yes	Yes

Where to Go Next

- Containers: `Bag`, `Vault`, `Store` and `Journal` one by one.
- Typed Readers: read values with the type you expect.
- Validation: schemas, when they are checked and what happens when they fail.
- Translations: the exceptions of the package and how to translate them.

Last updated on 15/11/2026

Containers

Containers

Anonymous · 15/11/2026

Containers

All the containers implement `Derafu\Container\Contract\ContainerInterface`, which has the methods to read and write by key (with dot notation), plus the typed reader:

```
use Derafu\Array\ArrayReader;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Criteria;

interface ContainerInterface extends ArrayAccess, JsonSerializable
{
    public function all(): array;
    public function set(string $key, mixed $value): static;
    public function get(string $key, mixed $default = null): mixed;
    public function has(string $key): bool;
    public function clear(?string $key = null): void;
    public function collection(): ArrayCollection;
    public function matching(Criteria $criteria): ArrayCollection;

    public function getReader(): ArrayReader;
    public function getContainer(string $key, ?array $default = null): ?static;
    public function requireContainer(string $key): static;
}
```

The `ArrayAccess` offsets use the same keys: `$container['user.name']` is the same as `$container->get('user.name')`.

Wherever a container is given data (the constructor of every container, and `replace()` and `merge()` of `Bag`), the data can be an array, an `ArrayObject` or another container of any kind. A container is copied with its data only: the new container does not have the schema of the other one, it is valid for its own, and a change in one of them does not change the other.

```
$options = new Vault(
    new Bag(['name' => 'Ada']),
    ['name' => ['types' => 'string'], 'debug' => ['types' => 'bool', 'default' =>
false]]
);

$options->all(); // ['debug' => false, 'name' => 'Ada']
```

Bag

A flexible container without restrictions. It is the one to use when you only need a place to keep data.

```
use Derafu\Container\Bag;

$bag = new Bag();
$bag->set('user.name', 'John');
$bag->set('user.email', 'john@example.com');

echo $bag->get('user.name'); // "John"
```

It adds three methods:

```
$bag->merge(['user' => ['age' => 30]]); // Recursive: keeps user.name and user.email.
$bag->replace(['other' => 1]);          // Replaces everything.
$bag->remove('other');                  // Removes a key.
```

Vault

A container with a simple schema. Each key of the schema says its types, whether it is required, its accepted values, its default and how to normalize it. The validation is made by `symfony/options-resolver`.

```
use Derafu\Container\Vault;

$vault = new Vault(
    ['user' => ['name' => 'John']],
    [
        'user' => [
            'types' => 'array',
            'required' => true,
            'schema' => [
                'name' => ['types' => 'string', 'required' => true],
                'age' => ['types' => 'int', 'default' => 18],
            ],
        ],
    ]
);

$vault->get('user.age'); // 18, the default.
```

The constructor takes the data, the schema and whether the keys that the schema does not define

are allowed (`false` by default). See Validation for every key of the schema and for what happens on each change.

Store

A container validated with JSON Schema (draft 2020-12) through `opis/json-schema`. Use it when the schema comes from outside, or when you need what JSON Schema has and the simple schema of `Vault` does not (formats, `items`, `oneOf`, patterns...).

```
use Derafu\Container\Store;

$store = new Store(
    ['user' => ['name' => 'John', 'email' => 'john@example.com']],
    [
        'properties' => [
            'user' => [
                'type' => 'object',
                'required' => ['name', 'email'],
                'properties' => [
                    'name' => ['type' => 'string'],
                    'email' => ['type' => 'string', 'format' => 'email'],
                ],
            ],
        ],
    ]
);
```

`$schema` and `type: object` are added to the schema when you do not give them, so `getSchema()` returns them. The `default` of a property is applied when the property has no value.

Journal

A sequential container: you add items at the end and read them newest first.

```
use Derafu\Container\Journal;

$journal = new Journal();
$journal->add(['level' => 'info', 'message' => 'System started']);
$journal->add(['level' => 'error', 'message' => 'Connection failed']);

$errors = array_filter(
    $journal->reverse(), // Newest first.
    fn (array $entry) => $entry['level'] === 'error'
);
```

Reading the Underlying Collection

`collection()` returns the `ArrayCollection` that holds the data, and `matching()` filters it with a Doctrine `Criteria`:

```
use Derafu\Container\Bag;
use Doctrine\Common\Collections\Criteria;

$users = new Bag([['name' => 'Ana', 'age' => 20], ['name' => 'Bo', 'age' => 10]]);

$adults = $users->matching(
    Criteria::create()->where(Criteria::expr()->gte('age', 18))
);
```

`collection()` is the live collection, not a copy. Changing it directly does not go through the validation of `Vault` and `Store`; see Validation.

Last updated on 15/11/2026

Typed Readers

Typed Readers

Anonymous · 15/11/2026

Typed Readers

Every container exposes `getReader(): Derafu\Array\ArrayReader`. Typed reads and explicit conversions use Derafu Array, whose types and conversion rules come from Derafu Value.

```
use Derafu\Container\Bag;

$bag = new Bag(['transport' => ['host' => 'localhost', 'port' => 993]]);
$reader = $bag->getReader();

$host = $reader->requireString('transport.host'); // 'localhost'
$port = $reader->requireInt('transport.port');    // 993
$ssl = $reader->getBool('transport.ssl', true);  // true
```

The typed getters belong to the reader. Container exposes storage, schema validation, Selector operations and creation of child containers.

Presence and Defaults

Optional reads return the default for missing or null values. Required reads reject both. A present value of the wrong type throws instead of using the default. Valid values such as `0`, `false`, `''` and `[]` are preserved.

Use `hasKey()` to distinguish a null value from an absent key and `hasValue()` to require a non-null value. Defaults are returned without validation or conversion.

```
$reader = (new Bag(['retries' => 0, 'note' => null]))->getReader();
$reader->getInt('retries', 3); // 0
$reader->hasKey('note');       // true
$reader->hasValue('note');     // false
$reader->getString('note', 'none'); // 'none'
```

Validation and Explicit Conversion

Strict getters check existing PHP types. For example, `getFloat()` requires a float; use `getAsFloat()` to convert an integer or numeric text. A conversion returns a value without changing the stored data.

```
$bag = new Bag(['port' => '993', 'count' => 2, 'enabled' => 'off']);
$reader = $bag->getReader();

$reader->requireAsInt('port');   // 993
$reader->getAsFloat('count');    // 2.0
$reader->requireAsBool('enabled'); // false
$bag->get('port');              // '993': unchanged
```

See Array Readers for the complete strict and conversion method pairs, and Value Casting for accepted conversions.

Collections, Enums and Objects

Collection element types are explicit ValueType instances. Arrays preserve arbitrary keys, maps require string keys and lists require consecutive integer keys starting at zero.

```
use Derafu\Value\ValueType;

$bag = new Bag(['ids' => [1, 2], 'amounts' => ['12.5', 2]]);
$reader = $bag->getReader();

$reader->requireListOf('ids', ValueType::int()); // [1, 2]
$reader->requireListOf('ids', ValueType::toFloat()); // [1.0, 2.0]
$reader->requireArrayOf('amounts', ValueType::toFloat()); // [12.5, 2.0]
```

```
enum Encryption: string
{
    case Ssl = 'ssl';
    case Tls = 'tls';
}

$mailer = new stdClass();
$reader = (new Bag(['encryption' => 'ssl', 'mailer' => $mailer]))->getReader();

$reader->require('encryption', ValueType::backedEnum(Encryption::class)); //
Encryption::Ssl
$reader->require('mailer', ValueType::instanceOf(stdClass::class)); // The same
object.
```

Generic reads and collection methods preserve their PHPStan generic return types. See Value Types for nullable types, unions, dates and custom types.

Navigation and Snapshots

The reader checks for a literal key matching the complete string first, then follows dot notation when

that key is absent. Numeric path segments can address array positions. For example, `requireInt('servers.0.port')` reads the port of the first server.

Container's `get()`, `set()`, `has()`, `clear()` and child-container methods retain the full Selector expression language, including bracket indexing. ArrayReader does not interpret those Selector expressions.

A reader is a snapshot of raw stored array data, independent of overrides of Container's `get()`. Schema defaults and normalizers have already been applied to that data. Obtain a new reader after mutations:

```
$bag = new Bag(['port' => 5432]);
$before = $bag->getReader();
$bag->set('port', 3306);

$before->requireInt('port');           // 5432
$bag->getReader()->requireInt('port'); // 3306
```

Objects inside a snapshot retain their identity; they are not cloned.

Nested Containers

`getContainer($key, $default = null)` and `requireContainer($key)` belong to Container. They read array data through Selector and create a child of the same container class. A required child rejects missing/null values; an optional child returns null or constructs a child from its supplied array default. Present values of the wrong type throw.

```
$bag = new Bag(['transports' => [['host' => 'localhost', 'port' => 993]]]);
$transport = $bag->requireContainer('transports[0]');

$transport->getReader()->requireString('host'); // 'localhost'
$transport->set('port', 995);
$bag->get('transports[0].port'); // 993: changing the child does not write back.
```

A nested reader provides typed access without constructing a mutable container; use `getReader()->requireReader('transports.0')` when that is all you need.

Child containers do not automatically inherit a parent's schema. Subclasses whose constructor needs additional context override `createChild(array $data, string $key): static`.

Exceptions

Reader failures follow the Array exception contract: missing/null required values raise Array MissingValueException and invalid types/conversions raise Array TypeMismatchException. Value type

mismatches retain their full path and original cause.

Child-container operations retain `Derafu\Container\Exception\MissingValueException` and `UnexpectedTypeException`. An invalid child value preserves the Value validation exception as its cause. Selector failures retain Selector's exception contract. See Translations for provider registration.

`get()` and ArrayAccess return `mixed`. Use `getReader()` when a typed value is required.

Last updated on 15/11/2026

Validation

Validation

Anonymous · 15/11/2026

Validation

`Vault` and `Store` have a schema, and they keep their data valid: a change that breaks the schema is rejected, and the container stays as it was. `Bag` and `Journal` have no schema and accept everything.

When the Data Is Checked

Operation	What is checked
The constructor	The data against the schema.
`set()` and `$container['key'] = ...`	The data that would result from the change.
`clear()` and `unset($container['key'])`	The data that would result from the change.
`setSchema()`	The stored data against the new schema.
`validate()`	The stored data against the current schema.

The data is also **normalized**: the defaults are applied and, in a `Vault`, the normalizers run. What is stored after a `set()` is the resolved data, as after the constructor.

```
use Derafu\Container\Vault;

$vault = new Vault(
    ['host' => 'imap.example.com'],
    [
        'host' => ['types' => 'string', 'required' => true],
        'port' => ['types' => 'int', 'default' => 993],
    ]
);

$vault->set('port', 'abc');
// InvalidOptionsException: The option "port" with value "abc" is expected to be
// of type "int", but is of type "string".

$vault->getReader()->requireInt('port'); // 993: nothing changed.
```

The change is made on a copy and the container takes it only if it is valid, so there is nothing to undo and nothing is silenced: the exception always reaches the caller.

Removing Keys

`clear()` is checked like any other change. Removing a key with a default gives it back its default; removing a required key is rejected:

```
$vault->clear('port');  // Fine: port is 993 again.
$vault->clear('host');  // MissingOptionsException: The required option "host" is
missing.
$vault->clear();        // The same: an empty container does not satisfy the schema.
```

Changing the Schema

`setSchema()` checks (and normalizes) the data that the container already has. If it does not comply, neither the schema nor the data change.

```
$vault = new Vault(['host' => 'a.example.com']);   // No schema: anything goes.

$vault->setSchema([
    'host' => ['types' => 'string', 'required' => true],
    'port' => ['types' => 'int', 'default' => 993],
]);

$vault->getReader()->requireInt('port'); // 993: the default was applied to the stored
data.
```

The Schema of a Vault

The schema is an array: each key is a field, and its value says how the field is validated. The keys of the field are:

Key	Meaning
`types`	A type, or an array of types, that the value can have: `string`, `int`, `float`, `bool`, `array`, `null`, `object`, a class name… (the types of `OptionsResolver`).
`required`	`true` when the field has to be present.
`choices`	The values that are accepted.
`default`	The value of the field when it is not given.
`normalizer`	A closure `fn (Options $options, mixed $value): mixed` that converts the value after it is validated.
`schema`	The schema of the fields inside an array field. It nests as deep as needed.

```
$vault = new Vault(
    ['price' => '12.50', 'status' => 'active', 'user' => ['name' => 'Ada']],
```

```
    [
        'price' => [
            'types' => ['string', 'float'],
            'required' => true,
            'normalizer' => fn ($options, $value) => (float) $value,
        ],
        'status' => ['types' => 'string', 'choices' => ['active', 'inactive']],
        'user' => [
            'types' => 'array',
            'schema' => [
                'name' => ['types' => 'string', 'required' => true],
                'age' => ['types' => 'int', 'default' => 18],
            ],
        ],
    ]
);

$vault->getReader()->requireFloat('price');   // 12.5
$vault->getReader()->requireInt('user.age');  // 18
$vault->set('status', 'other');  // Rejected: not one of the choices.
```

A field whose configuration is not valid (a `types` that is not a string or an array of strings, a `schema` that is not an array, a `normalizer` that is not a closure, a field that is not an array…) throws `TranslatableInvalidArgumentException` when the schema is used.

Keys the Schema Does Not Define

By default a key that is not in the schema is rejected:

```
$vault->set('endpoint', 'x');
// UndefinedOptionsException: The option "endpoint" does not exist.
```

The policy can be changed in three places, from the most general to the most specific:

1. The constructor: `new Vault($data, $schema, allowUndefinedKeys: true)`. The `Vault` keeps it and uses it in `set()`, `clear()`, `setSchema()` and `validate()`.
2. The key `__allowUndefinedKeys` in a schema (`true` or `false`), which overrides the constructor for that level.
3. `validate(true)` or `validate(false)`, only for that call.

A nested schema inherits the policy of the level above and can override it with its own `__allowUndefinedKeys`:

```
$vault = new Vault($data, [
    '__allowUndefinedKeys' => true,                // The root accepts more keys...
    'user' => [
        'schema' => [
```

```
            '__allowUndefinedKeys' => false,        // ...but `user` does not.
            'name' => ['types' => 'string'],
        ],
    ],
]);
```

The Exceptions of a Vault

The errors of the data are the exceptions of `symfony/options-resolver` (`InvalidOptionsException`, `MissingOptionsException`, `UndefinedOptionsException`), which extend `InvalidArgumentException`. They are not translated. The errors of the schema itself are `TranslatableInvalidArgumentException`.

The Schema of a Store

A `Store` is validated with JSON Schema (draft 2020-12). You can give the schema as an array; `$schema` and `type: object` are added if they are missing.

```
use Derafu\Container\Store;

$store = new Store(
    ['name' => 'Ada'],
    [
        'required' => ['name'],
        'properties' => [
            'name' => ['type' => 'string'],
            'lang' => ['type' => 'string', 'default' => 'es'],
        ],
    ]
);

$store->get('lang');                // "es": the default of the property.
$store->set('name', 5);             // Rejected.
$store->clear('name');              // Rejected: `name` is required.
```

The errors are `TranslatableInvalidArgumentException`, with every problem found and where it is:

```
Failed to validate the data against the JSON schema. The data (integer) must
match the type: string in /name.
```

JSON Schema defaults are applied to the data, but the data must be an object at its root: an empty container is validated as `{}`.

What Is Not Checked

The validation guards the methods of the container. These ways in do not go through it:

- `collection()` returns the live `ArrayCollection`. Changing it changes the container without validating. It is a deliberate way for the code that needs to work with the collection directly.
- An object stored in the container can be changed by whoever holds it.

`validate()` checks the stored data again, for when you used one of those:

```
$vault->collection()->set('port', 'abc');   // Not checked.
$vault->validate();                         // InvalidOptionsException.
```

`validate()` does not change the data. In a `Vault` it takes an optional argument to override the policy of undefined keys for that call.

Last updated on 15/11/2026

Translations

Translations

Anonymous · 15/11/2026

Translations

The exceptions of `derafu/container` can be translated. This page explains what is translatable, what ships out of the box and how to activate it in your application.

What’s Translatable

Every exception the package throws is translatable (see `derafu/translation`):

- `MissingValueException` and `UnexpectedTypeException`, from child-container creation through `getContainer()` / `requireContainer()`.
- The errors of a `Store` (data that does not comply with the JSON schema, data or schema that can not be converted to JSON).
- The errors of the schema of a `Vault`.
- A value that can not be used as the key of a container.

The errors of the data of a `Vault` are the exceptions of `symfony/options-resolver`, which are not translatable.

Messages are written in ICU MessageFormat, with the English text as the translation id and named placeholders:

```
use Derafu\Container\Bag;

try {
    (new Bag(['transport' => 'abc']))->requireContainer('transport');
} catch (\Throwable $e) {
    $exception = $e;
    echo $e->getMessage();
    // "The value of transport must be of type array; string given."
}
```

Without a translator configured, `getMessage()` already returns the formatted English text, so the package works as before whether or not you set up translation.

Reader and Selector Errors

Reads performed through `getReader()` execute in Derafu Array and use its translatable exceptions. Register Array's resource provider to translate those messages; see Array Translations. Direct Value operations use Value's provider, described in Value Translations.

Container retains its own exception classes for child creation. If the selected child value is not an array, UnexpectedTypeException preserves the Value mismatch as its previous exception. The outer Container message remains translated with Container's provider. Selector failures remain Selector exceptions.

What Ships

- `resources/translations/errors+intl-icu.es.php`: a Spanish translation for every message of the package.
- `Derafu\Container\Translation\ContainerTranslationResourceProvider`: a `TranslationResourceProviderInterface` pointing at that directory.
- `resources/config/container-services.yaml`: a recipe for applications that use `symfony/dependency-injection`.

None of them does anything by itself: `derafu/container` is a library, it does not build a `Translator`.

Activating It (Plain PHP)

```
use Derafu\Container\Translation\ContainerTranslationResourceProvider;
use Derafu\Translation\TranslatorFactory;

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

echo $exception->trans($translator);
// "El valor de transport debe ser de tipo array; se recibió string."
```

Activating It (Dependency Injection)

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

```
services:
    Symfony\Component\Translation\Translator:
        factory: ['Derafu\Translation\TranslatorFactory', 'create']
        arguments:
            $defaultLocale: 'es'
            $fallbackLocales: ['es', 'en']
            $resourceProviders: !tagged_iterator derafu_translation.resource_provider
```

`container-services.yaml` tags `ContainerTranslationResourceProvider` with `derafu_translation.resource_provider`, so it is picked up automatically.

Last updated on 15/11/2026