Docs

Repository Project

Entity repositories with in-memory data sources and explicit factories for custom storage implementations

Anonymous · 15/11/2026 · #php

Table of Contents

Repository Project 3
- ***Introduction*** 4
- ***Basic Usage*** 6
- ***Data Sources*** 9
- ***Entities and Mapping*** 13
- ***Querying*** 17
- ***Repository Factories*** 21
- ***Symfony Integration*** 24

Repository Project

Entity repositories with in-memory data sources and explicit factories for custom storage implementations

Anonymous · 15/11/2026 · #php

Derafu Repository

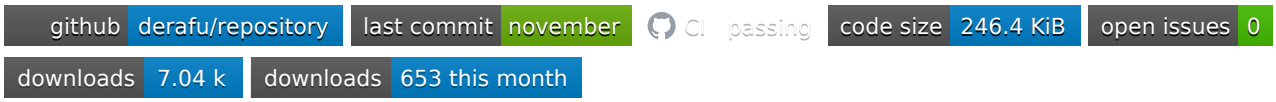
Last updated on 15/11/2026

Introduction

Entity repositories for PHP with file and enum sources, in-memory queries and custom repository factories

Anonymous · 15/11/2026

Lightweight Data Source Management for PHP



Derafu Repository provides an entity query contract, an in-memory repository and a manager that creates repositories through factories. Use the built-in data provider for catalogues and lookup tables, or register factories for repositories with their own infrastructure dependencies.

Installation

PHP 8.5 or later is required.

```
composer require derafu/repository
```

To read YAML files, also install the optional YAML component:

```
composer require symfony/yaml
```

Components

Component	Responsibility
`RepositoryInterface`	Entity queries, Doctrine `Criteria` matching, counting and the entity class name
`Repository`	In-memory implementation; also provides `all()` to read its raw rows
`CriteriaEvaluatorInterface`	Evaluate a Doctrine `Criteria` over rows with the semantics of a database
`DataSourceInterface`	Read a complete dataset as a PHP array
`RepositorySourceProviderInterface`	Contribute named origins from independent packages without loading them

Component	Responsibility
`ArrayRepositorySourceProvider`	Declare sources from a supplied map
`DataProvider`	Resolve sources, optionally cache raw data, normalize rows and retain loaded data
`DataProviderRepositoryFactory`	Construct in-memory repositories using data-provider sources and entity metadata
`RepositoryFactoryInterface`	Construct a repository using its own dependencies
`RepositoryManager`	Select a factory, create a repository on demand and reuse its instance

Choosing a Data Access Path

For a small or medium-sized catalogue, use a PHP, JSON or YAML file, a backed enum, or a custom data source. The provider reads the complete dataset; `Repository` filters and sorts it in memory.

For direct database or remote-service queries, register a repository factory. Its repository can execute each query against the backend and implement `RepositoryInterface` without extending the in-memory `Repository`. The package supplies the extension point; it does not supply a production SQL repository or an ORM.

Reusing a repository instance does not determine how fresh its results are. The built-in in-memory repository uses loaded data; a custom repository controls its own queries and caching.

Guides

- Basic usage: create a repository or use the manager.
- Data sources: files, enums, normalization, caching and extension points.
- Entities and mapping: entity classes, identifiers and specialized repositories.
- Querying: lookup, filtering, ordering and Doctrine Criteria.
- Repository factories: repositories with independent dependencies.
- Symfony integration: configure sources, factories and translation resources.

Last updated on 15/11/2026

Basic Usage

Create an in-memory repository and request repositories through RepositoryManager

Anonymous · 15/11/2026

Basic Usage

Create an In-Memory Repository

Pass the rows directly, keyed by their repository identifiers. String keys, numeric lists and sparse numeric keys are supported.

Each row is an array whose field names are strings. Numeric repository identifiers are valid; numeric field names inside a row raise `TypeMismatchException` when the repository loads its source. Field values, including nested arrays, are preserved without conversion. This applies to every entity class, including `stdClass`.

```
use Derafu\Repository\Entity;
use Derafu\Repository\Repository;

$rows = [
    'prod-001' => [
        'name' => 'Laptop XPS',
        'category' => 'computers',
        'price' => 1299.99,
    ],
    'prod-002' => [
        'name' => 'Keyboard',
        'category' => 'accessories',
        'price' => 59.90,
    ],
];

$repository = new Repository($rows, Entity::class, idAttribute: 'id');
$product = $repository->find('prod-001');
$name = $product?->getAttribute('name'); // 'Laptop XPS'.
$all = $repository->findAll();
$computers = $repository->findBy(['category' => 'computers']);
$laptop = $repository->findOneBy(['name' => 'Laptop XPS']);
$total = $repository->count(); // 2.
```

`idAttribute` fills a missing field from the row key. An existing field value is preserved. `find()` uses the row key, even if that field contains a different value.

The direct constructor accepts an array, supported array-access objects or a PHP file path returning rows. Each row must be an array. When no entity class is supplied, the direct `Repository` constructor returns `stdClass` objects.

Use the Manager

The manager delegates construction to an injected factory. A plain identifier such as `products` uses the default factory's entity class, `Derafu\Repository\Entity`. With Symfony, import the complete service configuration. Without a container, assemble the dependencies explicitly:

```
use Derafu\Repository\Factory\DataProviderRepositoryFactory;
use Derafu\Repository\Service\ArrayRepositorySourceProvider;
use Derafu\Repository\Service\CriteriaEvaluator;
use Derafu\Repository\Service\DataProvider;
use Derafu\Repository\Service\DataSource\DataSourceResolverRegistry;
use Derafu\Repository\Service\DataSource\EnumDataSourceResolver;
use Derafu\Repository\Service\DataSource\FileDataSourceResolver;
use Derafu\Repository\Service\DataSource\FileFormat\FileFormatReaderRegistry;
use Derafu\Repository\Service\DataSource\FileFormat\JsonFileFormatReader;
use Derafu\Repository\Service\DataSource\FileFormat\PhpFileFormatReader;
use Derafu\Repository\Service\DataSource\FileFormat\YamlFileFormatReader;
use Derafu\Repository\Service\RepositoryManager;

$readers = new FileFormatReaderRegistry([
    new PhpFileFormatReader(),
    new JsonFileFormatReader(),
    new YamlFileFormatReader(),
]);
$resolvers = new DataSourceResolverRegistry([
    new EnumDataSourceResolver(),
    new FileDataSourceResolver($readers),
]);
$provider = new DataProvider(
    sourceProviders: [new ArrayRepositorySourceProvider([
        'products' => '/path/to/products.json',
    ])],
    sourceResolvers: $resolvers,
);
$factory = new DataProviderRepositoryFactory($provider, new CriteriaEvaluator());
$manager = new RepositoryManager(defaultFactory: $factory, dataProvider: $provider);

$available = $manager->getAvailableRepositories(); // ['products'].
$products = $manager->getRepository('products');
$product = $products->find('prod-001');
```

The file contains the rows themselves, without a surrounding `products` property. Its extension selects the reader. See data sources for supported formats.

Listing available repositories does not read their data or construct them. A successful `getRepository()` call is cached by identifier: subsequent calls return the same repository instance.

Typed Repositories

`RepositoryInterface` and `Repository` are generic over their entity (`T`, any object). `getRepository()` types the repository it returns by its argument:

```
$products = $manager->getRepository(Product::class); // RepositoryInterface<Product>.
$product = $products->find('prod-001');              // Product|null.
$computers = $products->findBy(['category' => 'computers']); // Product[].

$statuses = $manager->getRepository('statuses');     // RepositoryInterface<object>.
```

With an entity class or an entity interface, the query methods return instances of it. With any other identifier the repository is typed with `object`.

The types are PHPDoc for static analysis tools: they do not change what a repository returns at run time. The `Repository` constructor takes its entity class as a `class-string<T>`.

Query Contract and Raw Rows

`RepositoryInterface` exposes entity queries. It does not require database persistence or a particular constructor. It extends Doctrine Collections' `Selectable`, so `matching()` accepts a `Criteria`.

The concrete `Repository` class also provides `all()`, which returns the stored rows keyed by identifier, whereas `findAll()` returns entity objects. The rows cannot be modified through the repository.

Depend explicitly on the concrete class when you need `all()`. A repository obtained from a custom factory is only required to implement the query contract.

Last updated on 15/11/2026

Data Sources

Built-in file and enum sources, custom sources, row normalization and caching

Anonymous · 15/11/2026

Data Sources

`DataProvider` receives an iterable of `RepositorySourceProviderInterface` implementations. Each provider returns a map of source identifiers to strings or `DataSourceInterface` instances. Duplicate identifiers are rejected. The manager uses these identifiers to resolve repositories.

Built-In Sources

Source	Configuration value	Requirements
PHP file	`/path/to/products.php`	File returns an array
JSON file	`/path/to/products.json`	Document decodes to an array using associative decoding
YAML file	`/path/to/products.yaml`	Document parses to an array; `symfony/yaml` is installed
Backed enum	`enum:App\Enum\Status`	Class is a string-backed or integer-backed enum

The built-in YAML reader recognizes `.yaml`. A raw string without the `enum:` prefix is treated as a file path. File reader selection uses the lowercased extension.

JSON and YAML objects or lists may describe the dataset. For example:

```
{
    "prod-001": {"name": "Laptop XPS", "price": 1299.99},
    "prod-002": {"name": "Keyboard", "price": 59.90}
}
```

Pass JSON and YAML paths through `DataProvider`; the direct `Repository` constructor treats a string as a PHP file to require.

Row Normalization

The provider converts a non-array entry into a row under `nameAttribute`, and fills missing identifiers under `idAttribute`. Defaults are `name` and `id`. Both settings must be strings.

```
use Derafu\Repository\Service\ArrayRepositorySourceProvider;
```

```
use Derafu\Repository\Service\DataProvider;

$provider = new DataProvider(
    sourceProviders: [new ArrayRepositorySourceProvider([
        'payment_methods' => '/path/to/payment_methods.php',
    ])],
    sourceResolvers: $resolvers,
    config: [
        'normalization' => [
            'idAttribute' => 'codigo',
            'nameAttribute' => 'glosa',
        ],
    ],
);
```

`$resolvers` is the explicitly assembled registry from basic usage, or the registry supplied by the service container.

For a PHP file returning `[1 => 'Cash', 2 => 'Transfer']`, the resulting rows are:

```
[
    1 => ['codigo' => 1, 'glosa' => 'Cash'],
    2 => ['codigo' => 2, 'glosa' => 'Transfer'],
];
```

Array rows preserve their fields and existing ID values. Source keys are preserved, including sparse numeric keys. The provider returns the normalized dataset as an `ArrayObject` and reuses it on subsequent `fetch()` calls for that identifier.

Enums

```
use Derafu\Repository\Service\ArrayRepositorySourceProvider;
use Derafu\Repository\Service\DataProvider;
use Derafu\Repository\Service\RepositoryManager;

enum Status: string
{
    case Active = 'active';
    case Inactive = 'inactive';
}

// $resolvers and $criteriaEvaluator are assembled application dependencies.
$provider = new DataProvider(
    [new ArrayRepositorySourceProvider(['statuses' => 'enum:' . Status::class])],
    $resolvers,
);
$factory = new \Derafu\Repository\Factory\DataProviderRepositoryFactory($provider,
```

```
$criteriaEvaluator);
$repository = (new RepositoryManager($factory, $provider))->getRepository('statuses');
$active = $repository->find('active');
```

The enum source emits `case value => case name`; the provider then normalizes it. Here the generic entity contains `id = 'active'` and `name = 'Active'`.

To return enum cases themselves, configure the enum class as the repository entity, as described in entities and mapping.

Custom Data Sources

Implement `DataSourceInterface::read(): array` and pass the instance directly. The following example reads a complete catalogue through an application-supplied callback:

```
use Derafu\Repository\Contract\DataSourceInterface;
use Derafu\Repository\Service\ArrayRepositorySourceProvider;
use Derafu\Repository\Service\DataProvider;

final class CatalogSource implements DataSourceInterface
{
    /** @param Closure(): array<array-key, mixed> $readCatalog */
    public function __construct(private readonly Closure $readCatalog)
    {
    }

    public function read(): array
    {
        return ($this->readCatalog)();
    }
}

$source = new CatalogSource(fn (): array => ['A' => ['name' => 'Alpha']]);
$provider = new DataProvider(
    [new ArrayRepositorySourceProvider(['catalog' => $source])],
    $resolvers,
);
```

A PDO-backed source is possible, but this contract reads the whole dataset before queries run in memory. Use a repository factory when queries should execute directly against the database.

For custom string schemes, implement `DataSourceResolverInterface` and supply a `DataSourceResolverRegistry`. Resolver order matters: the built-in file resolver accepts every string, so it must come last. Both registries require explicit iterables in their constructors; they do not instantiate built-in services internally. Registering after the catch-all cannot override it. The Symfony configuration supplies the built-in services and accumulates extensions through tags. Implement

`FileFormatReaderInterface` and provide a `FileFormatReaderRegistry` to support additional file formats.

Caching and Data Lifetime

The optional `$cache` constructor argument accepts a PSR-16 cache. It stores raw data before normalization. An array, including an empty array, is a hit; a non-array result is treated as a miss and the source is read again. Cache backend exceptions propagate.

Independently of that cache, the provider keeps loaded normalized datasets in memory. Updating a file or the external cache does not refresh a dataset already loaded by that provider instance. There is no refresh API on `DataProvider`.

Errors

Invalid source definitions, unsupported formats, non-backed enums and invalid normalization settings raise `DataProviderException`. An unreadable JSON file also raises that exception. PHP file execution, YAML parsing and application-supplied sources can additionally raise their own errors.

Last updated on 15/11/2026

Entities and Mapping

Entity classes, identifier conventions, mapping attributes and specialized in-memory repositories

Anonymous · 15/11/2026

Entities and Mapping

Entity Classes

`Repository` creates an entity from each selected row using the support package's hydration facilities. `stdClass` provides public fields; `Derafu\Repository\Entity` provides generic attributes and accessor methods. Application entity classes can provide their own properties and accessors.

The default data-provider factory's entity class is `Derafu\Repository\Entity`. The factory implements `ConfigurableInterface`; its configuration accepts `entityClass` and `repositoryClass` for the in-memory construction path:

```
use Derafu\Repository\Entity;
use Derafu\Repository\Repository;

$factory->setConfiguration([
    'entityClass' => Entity::class,
    'repositoryClass' => Repository::class,
]);
```

Repository Identifiers

For the default data-provider factory:

- An identifier containing a namespace separator is resolved as an entity class or entity interface convention.
- An identifier without a namespace separator uses the configured default entity class.
- An interface name ending in `Interface` is mapped from the `Contract` namespace segment to `Entity`, with the suffix removed. For example, `App\Contract\ProductInterface` maps to `App\Entity\Product`.

The source key must match the identifier passed to `getRepository()`. If the source is registered under `ProductInterface::class`, request that identifier rather than the concrete entity class.

Explicitly registered factories receive their identifier unchanged and do not use these entity naming conventions.

Specialized In-Memory Repositories

Associate an entity with a repository subclass through the `Mapping\Entity` attribute:

```
namespace App\Entity;

use App\Repository\ProductRepository;
use Derafu\Repository\Entity as GenericEntity;
use Derafu\Repository\Mapping as DEM;

#[DEM\Entity(repositoryClass: ProductRepository::class)]
class Product extends GenericEntity
{
}
```

The subclass can add domain-specific queries while retaining the in-memory implementation:

```
namespace App\Repository;

use App\Entity\Product;
use Derafu\Repository\Repository;

/**
 * @extends Repository<Product>
 */
class ProductRepository extends Repository
{
    /**
     * @return Product[]
     */
    public function findComputers(): array
    {
        return $this->findBy(['category' => 'computers']);
    }
}
```

`@extends Repository<Product>` states the entity of the subclass, so that static analysis types its query methods as `Product`. The entity class that the default factory passes to the constructor must be that same class.

Register `App\Entity\Product::class` as a data-provider source key. When requested, the default factory reads the attribute and constructs `ProductRepository` with the normalized data, entity class, ID attribute and injected criteria evaluator. Without a repository class in the attribute, the configured default repository class is used.

The selected class must be an instantiable `Repository` subclass compatible with those constructor arguments. For repositories requiring a connection or other construction dependencies, register a

factory instead.

The mapping attribute also exposes a `readOnly` metadata flag. The built-in manager and repository do not enforce it as a write restriction.

Enum Entities

A backed enum can be the entity class, causing query methods to return enum cases:

```
use Derafu\Repository\Repository;

$repository = new Repository(
    [
        'active' => ['id' => 'active'],
        'inactive' => ['id' => 'inactive'],
    ],
    Status::class,
    idAttribute: 'id',
);

$active = $repository->find('active'); // Status::Active.
```

Here `Status` is the backed enum from the data sources guide. The configured ID field must contain an integer or string accepted by the enum's `from()` method. A missing ID field is filled from the row key when `idAttribute` is configured. Invalid values and unsupported enum entities raise `EntityException`.

With a manager, register a namespaced enum class as the source key and use `'enum:' . Status::class` as its source value to combine enum reading with enum entities.

Generic Attribute Access

`Entity` supports `getAttribute()`, `setAttribute()`, `hasAttribute()`, `unsetAttribute()` and `toArray()`, along with magic properties and `getXyz()` / `setXyz()` accessors.

An explicitly stored null is an existing attribute. `unsetAttribute('name')` removes the attribute, whereas `unset($entity->name)` stores null under that name. Reading a missing attribute or invoking an unsupported method raises `EntityException`.

Typed Attribute Reading

`EntityInterface::getReader()` returns a `Derafu\Array\ArrayReader`. The default `Entity` implementation creates a snapshot of its attributes on each call. Changes made later through `setAttribute()` or `unsetAttribute()` appear in a new reader; existing readers retain their snapshot.

```
use Derafu\Repository\Entity;

$entity = new Entity();
$entity->setAttribute('id', 7)
    ->setAttribute('amount', '19.90')
    ->setAttribute('profile', ['name' => 'Ana'])
    ->setAttribute('optional', null);

$reader = $entity->getReader();
$id = $reader->requireInt('id');                 // 7.
$amount = $reader->requireAsFloat('amount');     // 19.9.
$name = $reader->requireString('profile.name');  // 'Ana'.
$exists = $reader->hasKey('optional');           // true.
$hasValue = $reader->hasValue('optional');       // false.
```

Strict methods such as `requireInt()` reject numeric strings; explicit conversion methods such as `requireAsFloat()` use the conversion rules of Derafu Value. Dot paths navigate nested arrays, with an existing literal key taking precedence over navigation. Reading does not change the stored attributes or reindex their arrays.

Reader failures use the exceptions from Derafu Array, while `getAttribute()` retains its `EntityException` contract. Custom classes that implement `EntityInterface` must also implement `getReader()`. See typed array readers for optional reads, required values, conversions and reader exceptions.

Last updated on 15/11/2026

Querying

Query methods, identifier lookup, filtering, sorting and Doctrine Criteria

Anonymous · 15/11/2026

Querying

`RepositoryInterface` defines the following operations. Examples in this guide use the in-memory `Repository`; custom implementations execute their own backend queries.

Method	Result
`find($id)`	Entity object or null
`findAll()`	Array of entity objects
`findBy($criteria, $orderBy, $limit, $offset)`	Array of matching entity objects
`findOneBy($criteria, $orderBy)`	First matching entity or null
`count($criteria)`	Number of matching rows
`matching($criteria)`	Collection of entity objects selected with Doctrine Criteria
`findByCriteria($criteria)`	The same entity objects as an array
`getClassName()`	Entity class name

In a repository typed with an entity (`RepositoryInterface<Product>`), “entity object” is an instance of that entity. See typed repositories.

Identifier Lookup

```
$product = $repository->find('prod-001');
$missing = $repository->find('unknown'); // null.
```

The in-memory implementation accepts string and integer identifiers and looks up the stored row key. Its optional `find()` lock arguments are unused. An `id` field inside the row does not change the lookup key.

Filtering and Counting

```
$computers = $repository->findBy(['category' => 'computers']);
$selected = $repository->findBy([
```

```
    'category' => ['computers', 'accessories'],
]);
$first = $repository->findOneBy(['category' => 'computers']);
$total = $repository->count();
$computerCount = $repository->count(['category' => 'computers']);
```

Fields in a criteria array are combined with AND. An array value means any of the listed values may match that field. The in-memory implementation compares values with PHP's non-strict `in_array()`.

A row that does not have a field has it as null. A null value matches the rows whose field is null, and a list that contains null matches them too. Any other value never matches a null field, so `0`, `''` and `false` are not null.

A field that no row has is rejected with a translatable invalid-argument exception. A repository without rows has no fields to check and accepts any field.

An empty criteria array matches every row. `findAll()` and the filtering methods return lists of entities rather than preserving source keys.

Ordering and Pagination

```
$page = $repository->findBy(
    criteria: ['category' => 'computers'],
    orderBy: ['price' => 'DESC', 'name' => 'ASC'],
    limit: 20,
    offset: 40,
);
```

The in-memory implementation processes ordering fields in the supplied order. A direction is `ASC` or `DESC`, in any case and with spaces around it. Any other direction, a direction that is not a string, or a field that no row has raises a translatable invalid-argument exception.

A null or missing field is the smallest value: it goes first when ascending and last when descending. Rows that tie keep their relative order.

An offset or limit below zero raises a translatable invalid-argument exception.

Doctrine Criteria

```
use Doctrine\Common\Collections\Criteria;
use Doctrine\Common\Collections\Order;

$expensiveProducts = $repository->matching(
    Criteria::create()
        ->where(Criteria::expr()->gt('price', 1000))
```

```
        ->orderBy(['price' => Order::Descending])
        ->setMaxResults(10),
);
```

`RepositoryInterface` extends Doctrine Collections' `Selectable`. `matching()` returns a collection of entity objects whose keys are their positions in the result, and `findByCriteria()` returns the same objects as an array.

For `Repository`, the criteria are evaluated over the stored rows before entities are created, with the semantics of a database:

- A missing field is null.
- A comparison with a null value is not true. `eq(field, null)` and `isNull()` ask for null, and `neq(field, null)` and `isNotNull()` ask for not null.
- `not()` leaves out the rows whose answer is unknown, as `neq()` and `notIn()` leave out null fields.
- `in()` with a list that contains null also matches null fields.
- The text operators `contains()`, `startsWith()` and `endsWith()` ignore the case of ASCII letters. The other letters must be the same.
- `memberOf()` looks inside a field that holds a list.
- A field with dots, such as `address.city`, reads inside the arrays of the row.
- A maximum of zero results is an empty result.

The supported operators are `eq`, `neq`, `lt`, `lte`, `gt`, `gte`, `in`, `notIn`, `contains`, `startsWith`, `endsWith`, `memberOf`, `isNull`, `isNotNull` and the compositions `andX`, `orX` and `not`. A field that no row has, or an operator that is not supported, raises a translatable invalid-argument exception.

The evaluation is implemented by `CriteriaEvaluator` through `CriteriaEvaluatorInterface`. `Repository` accepts another evaluator as its last constructor argument.

A custom SQL repository is responsible for implementing these query methods and translating supported criteria to its backend. The manager does not translate expressions or issue SQL.

Differences from a Database

The package tests the in-memory `Repository` against Doctrine's repository over a SQLite table with the same data. They answer alike except where a repository over rows has no schema or deliberately differs:

- The fields of a repository are the keys of its rows. A repository without rows accepts any field, and a field that no row has is unknown even if the entity class declares it.
- The text of `contains()`, `startsWith()` and `endsWith()` is searched as it is. The characters `%` and `_` are not wildcards.
- A limit below zero raises an exception, whereas SQLite takes it as no limit.

Raw Rows and Entities

`Repository::all()` returns raw rows, while `Repository::findAll()` returns entities. The rows of a repository cannot be modified through it, and it does not write to a source file or database. Each row must be an array.

Last updated on 15/11/2026

Repository Factories

Construct custom repositories with dependency injection and combine them with in-memory sources

Anonymous · 15/11/2026

Repository Factories

Register a factory when a repository requires its own connection, HTTP client or other infrastructure dependencies. The factory handles construction; the repository handles queries; the manager handles selection and instance reuse.

Factory Contract

```
namespace Derafu\Repository\Contract;

interface RepositoryFactoryInterface
{
    /**
     * @return RepositoryInterface<object>
     */
    public function create(string $repository): RepositoryInterface;
}
```

The identifier may be an entity class, an interface or an application-defined string. The factory returns an implementation of `RepositoryInterface`; the manager types the repository it returns by the requested entity class. Its repository does not need to extend `Repository` or accept the in-memory constructor arguments.

Application Factory

For an application that provides a `ProductRepository` implementation accepting a PDO connection:

```
namespace App\Repository;

use App\Entity\Product;
use Derafu\Repository\Contract\RepositoryFactoryInterface;
use Derafu\Repository\Contract\RepositoryInterface;
use PDO;

final class ProductRepositoryFactory implements RepositoryFactoryInterface
{
    public function __construct(private readonly PDO $connection)
```

```
    {
    }

    /**
     * @return RepositoryInterface<Product>
     */
    public function create(string $repository): RepositoryInterface
    {
        return new ProductRepository($this->connection);
    }
}
```

`ProductRepository` belongs to the application and implements the query contract, including `matching()`. It declares its entity with `@implements RepositoryInterface<Product>`. The package does not provide that class or a production SQL adapter. The application controls SQL, hydration, supported criteria and backend error handling.

Registration

```
use App\Entity\Product;
use App\Repository\ProductRepositoryFactory;
use Derafu\Repository\Service\RepositoryManager;

$manager = new RepositoryManager(
    factories: [
        Product::class => new ProductRepositoryFactory($connection),
    ],
);

$products = $manager->getRepository(Product::class);
$product = $products->find(123);
```

`$connection` is an application-provided PDO instance. A manager using only explicit factories does not need a data provider. The registered identifier is passed unchanged to `create()`; the manager does not resolve an entity class or inspect mapping attributes for it.

One factory instance can be registered under several identifiers. The manager caches successful repositories separately for each identifier. The factory decides how to interpret each ID.

Combining Factories and Data Sources

```
use App\Entity\Product;
use App\Repository\ProductRepositoryFactory;
use Derafu\Repository\Factory\DataProviderRepositoryFactory;
use Derafu\Repository\Service\ArrayRepositorySourceProvider;
use Derafu\Repository\Service\DataProvider;
```

```
use Derafu\Repository\Service\RepositoryManager;

$provider = new DataProvider(
    [new ArrayRepositorySourceProvider(['countries' => '/path/to/countries.php'])],
    $resolvers,
);
$defaultFactory = new DataProviderRepositoryFactory($provider, $criteriaEvaluator);
$manager = new RepositoryManager(
    defaultFactory: $defaultFactory,
    dataProvider: $provider,
    factories: [Product::class => new ProductRepositoryFactory($connection)],
);

$countries = $manager->getRepository('countries');
$products = $manager->getRepository(Product::class);
```

`$resolvers` and `$criteriaEvaluator` are the dependencies assembled in basic usage, or obtained from the service container.

The manager resolves requests in this order:

1. Return the instance already cached for the identifier.
2. Use the factory explicitly registered under that identifier.
3. Use the injected default factory, whose configuration belongs to the factory itself.

An explicit registration takes precedence even when the data provider declares the same identifier. Its construction does not read that data-provider source. With no registered factory and no default factory, the manager raises `ManagerException`.

`getAvailableRepositories()` returns the data-provider identifiers followed by explicit registration identifiers, without duplicates. It does not construct repositories or read source data.

Lifetime and Errors

Only successful constructions are cached. If a factory fails, a later request can retry it. Factories are not responsible for the manager's instance cache, and the manager does not refresh data or add result caching to a custom repository.

`ManagerException` instances propagate unchanged so their translation information is retained. Other exceptions raised during construction are exposed through a `ManagerException` containing their message. Exceptions raised by later repository queries are handled by the repository or its caller.

Symfony Integration

Wire file-backed catalogues, custom factories and translated exceptions with Symfony DependencyInjection

Anonymous · 15/11/2026

Symfony Integration

Import the package configuration to register the manager, data provider, default repository factory, criteria evaluator, source resolvers, file readers and translation resource provider. Symfony DependencyInjection, Config and YAML must be installed by the consuming application.

Import and Configure

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

parameters:
    derafu.repository.sources:
        App\Entity\Country: '%kernel.project_dir%/resources/data/countries.php'
        App\Contract\DocumentTypeInterface:
'%kernel.project_dir%/resources/data/document_types.php'
    derafu.repository.data_provider.config:
        normalization:
            idAttribute: codigo
            nameAttribute: glosa
    derafu.repository.repository_factory.config:
        entityClass: Derafu\Repository\Entity
        repositoryClass: Derafu\Repository\Repository
```

All three parameters default to empty maps. The provider defaults to `id` and `name` normalization attributes; the factory defaults to `Entity` and `Repository`. Configure only the values your application needs. Inject `RepositoryManagerInterface` into application services and call `getRepository()` with a source identifier. The interface identifier convention is described in entities and mapping.

Accumulate Sources from Independent Packages

A parameter map is convenient for one application’s sources. When several packages contribute sources, each package declares a tagged `RepositorySourceProviderInterface` implementation. Their maps are accumulated, so an extension package does not need to copy its dependency’s

entries.

```
namespace App\Repository;

use Derafu\Repository\Contract\RepositorySourceProviderInterface;

final readonly class CoreRepositorySourceProvider implements
RepositorySourceProviderInterface
{
    public function __construct(private string $dataDirectory)
    {
    }

    public function getSources(): iterable
    {
        return [
            \App\Entity\Country::class => $this->dataDirectory . '/countries.php',
            'statuses' => 'enum:App\\Enum\\Status',
        ];
    }
}
```

```
services:
    App\Repository\CoreRepositorySourceProvider:
        arguments:
            $dataDirectory: '%kernel.project_dir%/resources/data/repository'
        tags: ['derafu_repository.source_provider']

    Pro\Repository\ProRepositorySourceProvider:
        arguments:
            $dataDirectory: '%pro.project_dir%/resources/data/repository'
        tags: ['derafu_repository.source_provider']
```

`ProRepositorySourceProvider` implements the same contract and returns only Pro's entries; `%pro.project_dir%` is supplied by that package or application. The built-in `ArrayRepositorySourceProvider` contributes `derafu.repository.sources` through the same tag. Duplicate identifiers across providers, including that parameter map, raise a translatable `DataProviderException`; providers do not silently override each other.

Values may be file paths, `enum:` strings, other strings handled by a registered resolver, or `DataSourceInterface` objects injected into the provider. Providers declare origins without reading them. A data source reads a complete dataset; use a repository factory for queries executed directly against a database or remote service.

Add a Repository Factory

```
services:
    App\Repository\ProductRepositoryFactory:
        arguments:
            $connection: '@app.database_connection'
        tags:
            - { name: derafu_repository.repository_factory, repository:
'App\Entity\Product' }
```

The factory implements `RepositoryFactoryInterface`. Its tag's `repository` attribute selects the identifier passed to `getRepository()`. Add several tags to register the same factory for several identifiers. An explicitly tagged factory takes precedence over the default data-provider factory, even if a source with the same identifier exists. The manager reuses each successfully created repository.

Extend Resolvers and File Readers

```
services:
    App\Repository\ApiSourceResolver:
        arguments:
            $client: '@app.api_client'
        tags:
            - { name: derafu_repository.source_resolver, priority: 10 }

    App\Repository\CsvFileFormatReader:
        tags: ['derafu_repository.file_format_reader']
```

Resolvers implement `DataSourceResolverInterface`; readers implement `FileFormatReaderInterface`. Higher tag priorities run first. The built-in enum resolver has priority `100`; the catch-all file resolver has priority `-1000`. An ordinary custom resolver therefore runs before the file fallback. Readers select the first service that supports the extension; their tag also supports priority.

Criteria Evaluator and Cache

The default factory receives `CriteriaEvaluatorInterface` from the container and passes it to each in-memory repository. A deliberate application alias or decorator can customize that dependency. Runtime repositories and entities are constructed on demand, rather than registered as shared services.

The data provider's optional PSR-16 cache defaults to null. To enable it, override its `$cache` argument with the application's cache service while retaining the remaining package wiring.

Translated Exception Messages

The import tags `RepositoryTranslationResourceProvider` as `derafu_translation.resource_provider`. The application's translator must consume that collection, as described in the translation documentation. For example, when the application provides its own translator:

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

Without a service container, provide the resource provider directly:

```
use Derafu\Repository\Translation\RepositoryTranslationResourceProvider;
use Derafu\Translation\TranslatorFactory;

$translator = TranslatorFactory::create(
    'es',
    resourceProviders: [new RepositoryTranslationResourceProvider()],
);
```

Package exceptions expose `trans($translator)`. Their original messages are in English, and the package includes a Spanish catalogue.

Last updated on 15/11/2026