

Entities and Mapping

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

Anonymous · 20/10/2026

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 manager's default entity class is `Derafu\Repository\Entity`. Its configuration accepts `entityClass` and `repositoryClass` for the in-memory construction path:

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

$manager->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 Derafu\Repository\Repository;

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

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 and ID attribute. 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`.

Last updated on 20/10/2026