

Docs » Data Handling Category » Repository Project » Repository Factories

Repository Factories

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

Anonymous · 20/10/2026

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
{
    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`. Its repository does not need to extend `Repository`, expose container operations 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 Derafu\Repository\Contract\RepositoryFactoryInterface;
use Derafu\Repository\Contract\RepositoryInterface;
use PDO;

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

    public function create(string $repository): RepositoryInterface
```

```
{
    return new ProductRepository($this->connection);
}
```

`ProductRepository` belongs to the application and implements the query contract. 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\Service\DataProvider;
use Derafu\Repository\Service\RepositoryManager;

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

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

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 `DataProviderRepositoryFactory` with the configured data provider and manager settings.

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 data provider, 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.

Last updated on 20/10/2026