

Data Sources

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

Anonymous · 20/10/2026

Data Sources

`DataProvider` accepts a map of source identifiers to strings or `DataSourceInterface` instances. The manager uses these identifiers to resolve repositories.

Built-In Sources

Source	Configuration value	Requirements
PHP file	<code>/path/to/products.php</code>	File returns an array
JSON file	<code>/path/to/products.json</code>	Document decodes to an array using associative decoding
YAML file	<code>/path/to/products.yaml</code>	Document parses to an array; <code>symfony/yaml</code> is installed
Backed enum	<code>enum:App\Enum\Status</code>	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\DataProvider;
```

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

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

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

// A plain identifier selects the manager's generic entity class.
$provider = new DataProvider(['statuses' => 'enum:' . Status::class]);
$repository = (new RepositoryManager($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\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(['catalog' => $source]);
```

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. Providing a resolver iterable replaces the default resolver list; registering after the catch-all cannot override it. 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 20/10/2026