Resource Providers

Resource Providers

Anonymous · 09/09/2026

Resource Providers

This guide explains `TranslationResourceProviderInterface` — the piece that lets you (or a dependency injection container) discover translation directories from multiple sources — and when you'd write a custom `Symfony\Component\Translation\Loader\LoaderInterface` instead.

What a Resource Provider Is (and Isn't)

`TranslationResourceProviderInterface` declares **directories containing translation files**. It does not read or parse anything itself — that's `TranslationResourceRegistrar`'s job (see the API Reference), using `symfony/translation`'s own file loaders (YAML, JSON, PHP, XLIFF, PO, MO, CSV, INI).

```
interface TranslationResourceProviderInterface
{
    /**
     * @return iterable<string> Absolute paths to directories containing
     * translation files, named `domain(+intl-icu)?.locale.format`.
     */
    public function getDirectories(): iterable;
}
```

If your translations always live in files on disk, this is the interface to implement. If they come from somewhere else entirely (a database, an API, Redis), see Non-File Sources below — that's a different extension point.

The Built-In Implementation

For the common case — "I have N directories and that's it" — `SimpleTranslationResourceProvider` already does the job:

```
use Derafu\Translation\SimpleTranslationResourceProvider;

$provider = new SimpleTranslationResourceProvider([
    __DIR__ . '/translations',
    __DIR__ . '/vendor/some-package/translations',
```

```
]);
```

Writing Your Own

A custom provider is useful when the set of directories isn't a fixed, hardcoded list — for example, when it needs to be computed:

```
use Derafu\Translation\Contract\TranslationResourceProviderInterface;

final class PackageTranslationResourceProvider implements
TranslationResourceProviderInterface
{
    public function getDirectories(): iterable
    {
        // Every installed vendor package that ships its own translations,
        // discovered instead of hardcoded.
        foreach (glob(__DIR__ . '/../../vendor/*/*/resources/translations',
GLOB_ONLYDIR) as $dir) {
            yield $dir;
        }
    }
}
```

This is exactly the pattern used to expose a library's own bundled translations to whatever application consumes it: the library ships a small provider class pointing at its own `resources/translations` directory, and the consuming application registers it (directly, or via a tagged dependency-injection collection — see Advanced Usage).

```
use Derafu\Translation\Contract\TranslationResourceProviderInterface;

final class MyLibraryTranslationResourceProvider implements
TranslationResourceProviderInterface
{
    public function getDirectories(): iterable
    {
        return [__DIR__ . '/../../resources/translations'];
    }
}
```

Registering Providers

`TranslationResourceRegistrar::registerFromProviders()` accepts any `iterable` of `TranslationResourceProviderInterface` instances:

```
use Derafu\Translation\TranslationResourceRegistrar;

$registrar = new TranslationResourceRegistrar($translator);
$registrar->registerFromProviders([
    new SimpleTranslationResourceProvider([__DIR__ . '/translations']),
    new PackageTranslationResourceProvider(),
]);
```

Or pass them straight to `TranslatorFactory::create()`, which registers them as part of building the translator (see the API Reference for why that matters):

```
use Derafu\Translation\TranslatorFactory;

$translator = TranslatorFactory::create(
    defaultLocale: 'es',
    resourceProviders: [
        new SimpleTranslationResourceProvider([__DIR__ . '/translations']),
        new PackageTranslationResourceProvider(),
    ],
);
```

Registration order is precedence order: when two directories define the same key for the same domain/locale, the **last one registered wins**. If you want your own application's translations to be able to override a dependency's, register the dependency's provider(s) first and your own last.

Non-File Sources

If your translations genuinely don't live in files (a database table, a remote API, Redis), `TranslationResourceProviderInterface` isn't the right extension point — it only ever produces *directories*. Instead, write a `Symfony\Component\Translation\Loader\LoaderInterface` and register it directly on the `Translator`:

```
use Symfony\Component\Translation\Loader\LoaderInterface;
use Symfony\Component\Translation\MessageCatalogue;

final class DatabaseLoader implements LoaderInterface
{
    public function __construct(private readonly PDO $db)
    {
    }

    public function load(mixed $resource, string $locale, string $domain =
'messages'): MessageCatalogue
    {
        $stmt = $this->db->prepare(
            'SELECT message_key, message_text FROM translations WHERE locale = ? AND
```

```
domain = ?'
        );
        $stmt->execute([$locale, $domain]);

        $catalogue = new MessageCatalogue($locale);
        $catalogue->add($stmt->fetchAll(PDO::FETCH_KEY_PAIR), $domain);

        return $catalogue;
    }
}
```

```
$translator->addLoader('database', new DatabaseLoader($pdo));
$translator->addResource('database', 'irrelevant-for-this-loader', 'es', 'errors+intl-
icu');
```

This bypasses `TranslationResourceRegistrar` entirely — it's a direct use of `symfony/translation`'s own extension point, which this library builds on rather than replaces.

Last updated on 09/09/2026