# Advanced Usage

Advanced Usage Guide

Anonymous · 09/09/2026

# Advanced Usage Guide

This guide covers less common, but still fully supported, patterns.

## Reusable Message Templates

`TranslatableMessage` instances are plain value objects — build them once and reuse them wherever a `TranslatableInterface` is expected (including as an exception's message):

```
use Derafu\Translation\Contract\TranslatableInterface;
use Derafu\Translation\TranslatableMessage;

final class Messages
{
    public static function required(string $field): TranslatableInterface
    {
        return new TranslatableMessage('The field {field} is required.', ['field' =>
$field]);
    }

    public static function invalid(string $field, mixed $value): TranslatableInterface
    {
        return new TranslatableMessage(
            'The value {value} for field {field} is invalid.',
            ['field' => $field, 'value' => (string) $value],
        );
    }
}

throw new ValidationException(Messages::required('email'));
```

## Multiple Directories and Precedence

`TranslationResourceRegistrar` accepts a single directory, a list of directories, or a collection of resource providers — and registration order defines precedence:

```
use Derafu\Translation\TranslationResourceRegistrar;

$registrar = new TranslationResourceRegistrar($translator);
$registrar->registerDirectories([
    __DIR__ . '/vendor/some-dependency/translations',
    __DIR__ . '/translations', // Your own, registered last: it can override the
dependency's keys.
]);

$registrar->getRegisteredLocales(); // e.g. ['en', 'es']
$registrar->getRegisteredDomains(); // e.g. ['errors+intl-icu']
```

Prefer feeding providers straight into `TranslatorFactory::create()` instead of instantiating `TranslationResourceRegistrar` yourself whenever you can — see the API Reference for why registration needs to happen there rather than through a separately fetched registrar object.

## Dependency Injection (`symfony/dependency-injection`)

`derafu/translation` doesn't depend on `symfony/dependency-injection` (or any framework) — this is entirely optional wiring for applications that choose to use it. The package ships `resources/config/translation-services.yaml` with one thing: an alias from the translation-contracts interface to the concrete `Translator` service, so that code type-hinting the interface (as it should) still resolves correctly:

```
services:
    Symfony\Contracts\Translation\TranslatorInterface:
'@Symfony\Component\Translation\Translator'
```

Import it, then register your own `Translator` and any `TranslationResourceProviderInterface` implementations, tagged so they can be collected automatically:

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

services:
    App\Translation\MyTranslationResourceProvider:
        tags: ['derafu_translation.resource_provider']

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

Note the tag is applied explicitly on each provider service, rather than relying on Symfony's `_instanceof` autoconfiguration: `_instanceof` (like `_defaults`) only applies to services declared in the *same* YAML file that declares it — it does not propagate to services declared in a different file, even one that imports it.

## Decorating the Translator

Since `derafu/translation` builds directly on `Symfony\Contracts\Translation\TranslatorInterface`, any standard decorator pattern for that interface works unchanged — for example, logging every translation lookup:

```
use Psr\Log\LoggerInterface;
use Symfony\Contracts\Translation\TranslatorInterface;

final class LoggingTranslator implements TranslatorInterface
{
    public function __construct(
        private readonly TranslatorInterface $translator,
        private readonly LoggerInterface $logger,
    ) {
    }

    public function trans(string $id, array $parameters = [], ?string $domain = null,
?string $locale = null): string
    {
        $result = $this->translator->trans($id, $parameters, $domain, $locale);

        $this->logger->debug('Translation performed.', [
            'id' => $id,
            'domain' => $domain,
            'locale' => $locale,
            'result' => $result,
        ]);

        return $result;
    }

    public function getLocale(): string
    {
        return $this->translator->getLocale();
    }
}
```

Wrap the `Translator` built by `TranslatorFactory::create()` with your decorator, and pass the decorator — not the raw translator — to `trans()` calls and to anything else that needs a `TranslatorInterface`:

```
$translator = new LoggingTranslator(TranslatorFactory::create('en'), $logger);

echo $translator->trans('Hello {name}', ['name' => 'John'], 'messages+intl-icu');
// Logs the lookup, then: "Hello John"
```

Remember the `+intl-icu` domain suffix here too — the decorator forwards whatever domain it's given straight to the wrapped translator, so the same ICU rules apply.

Last updated on 09/09/2026