Quick Start

Quick Start Guide

Anonymous · 09/09/2026

Quick Start Guide

Get started with `derafu/translation` in a few minutes.

Installation

```
composer require derafu/translation
```

That's it — there are no optional dependencies to add separately. `symfony/translation`, `symfony/translation-contracts`, `symfony/yaml` and the `intl` PHP extension all come as hard requirements, so every built-in file format (YAML, JSON, PHP, XLIFF, PO, MO, CSV, INI) and ICU formatting work out of the box.

Basic Setup

1. Make an Exception Translatable

Extend one of the built-in translatable exceptions:

```
use Derafu\Translation\Exception\Core\TranslatableException;

class ValidationException extends TranslatableException
{
    // No additional code needed.
}
```

If you can't change the parent class (you already extend something else), use the trait instead:

```
use Derafu\Translation\Contract\TranslatableInterface;
use Derafu\Translation\Trait\TranslatableExceptionTrait;
use DomainException;

class ValidationException extends DomainException implements TranslatableInterface
{
    use TranslatableExceptionTrait;
```

```
}
```

See Exceptions for the full hierarchy.

2. Throw It

```
// 1. A plain string: used as-is, and as the translation id.
throw new ValidationException('Email is required.');

// 2. An array: first element is the message/id, the rest are named ICU
//    parameters.
throw new ValidationException([
    'The value {value} must be between {min} and {max}.',
    'value' => 42,
    'min' => 1,
    'max' => 10,
]);
```

`getMessage()` already returns the fully formatted English text at this point — no translator needed yet:

```
try {
    // ...
} catch (ValidationException $e) {
    echo $e->getMessage();
    // "The value 42 must be between 1 and 10."
}
```

3. Create Translation Files

Files are flat inside a directory and follow the naming convention `{domain}(+intl-icu)?.{locale}.{format}`. `TranslatableExceptionTrait` defaults to the `errors` domain, so add `+intl-icu` to activate ICU formatting for it:

```
// translations/errors+intl-icu.en.php
return [
    'The value {value} must be between {min} and {max}.' =>
        'The value {value} must be between {min} and {max}.',
];
```

```
// translations/errors+intl-icu.es.php
return [
    'The value {value} must be between {min} and {max}.' =>
        'El valor {value} debe estar entre {min} y {max}.',
];
```

YAML and JSON work exactly the same way:

```
# translations/errors+intl-icu.es.yaml
'The value {value} must be between {min} and {max}.': 'El valor {value} debe estar
entre {min} y {max}.'
```

```
// translations/errors+intl-icu.es.json
{
    "The value {value} must be between {min} and {max}.": "El valor {value} debe estar
entre {min} y {max}."
}
```

4. Build a Translator and Register the Files

```
use Derafu\Translation\TranslationResourceRegistrar;
use Derafu\Translation\TranslatorFactory;

// Builds a Symfony Translator with every supported loader already wired.
$translator = TranslatorFactory::create(
    defaultLocale: 'es',
    fallbackLocales: ['es', 'en'],
);

// Discovers every translation file in the directory and registers it.
$registrar = new TranslationResourceRegistrar($translator);
$registrar->registerDirectory(__DIR__ . '/translations');
```

5. Translate

```
try {
    // ...
} catch (ValidationException $e) {
    // Untranslated (English, formatted eagerly at construction time).
    echo $e->getMessage();

    // Translated to the translator's current locale.
    echo $e->trans($translator);

    // Translated to a specific locale.
    echo $e->trans($translator, 'es');
}
```

Common Use Cases

Form-Style Validation

```
class UserValidator
{
    public function validateEmail(string $email): void
    {
        if (empty($email)) {
            throw new ValidationException('The field {field} is required.', ['field'
=> 'email']);
        }

        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new ValidationException([
                'The value {value} is not a valid email.',
                'value' => $email,
            ]);
        }
    }
}
```

API Error Responses

```
use Derafu\Translation\Contract\TranslatableInterface;
use Symfony\Contracts\Translation\TranslatorInterface;
use Throwable;

class ApiErrorHandler
{
    public function __construct(
        private readonly TranslatorInterface $translator,
    ) {
    }

    public function handle(Throwable $e): array
    {
        $message = $e instanceof TranslatableInterface
            ? $e->trans($this->translator)
            : $e->getMessage();

        return ['error' => ['message' => $message]];
    }
}
```

Complex Messages (Pluralization, Gender)

```
throw new ValidationException(
    '{count, plural, =0{No files uploaded} one{1 file uploaded} other{# files
uploaded}}',
```

```
    ['count' => $fileCount],
);

throw new ValidationException(
    '{gender, select, female{She} male{He} other{They}} uploaded {count} files.',
    ['gender' => $user->getGender(), 'count' => $fileCount],
);
```

Next Steps

1. Read about ICU Message Format for pluralization, gender selection and more.
2. Learn about Resource Providers for multi-directory and multi-package setups.
3. Check the API Reference for every class and interface.
4. See Advanced Usage for composition and DI wiring.

Common Pitfalls

1. Always include the `other` case in plural/select patterns:

   ```
   // Wrong.
   '{gender, select, female{She} male{He}}'

   // Correct.
   '{gender, select, female{She} male{He} other{They}}'
   ```

2. Provide every parameter the message uses:

   ```
   // Missing 'field' — the {field} placeholder is left unexpanded.
   throw new ValidationException('The field {field} is required.');
   // getMessage(): "The field {field} is required."

   // Correct.
   throw new ValidationException('The field {field} is required.', ['field' =>
   'email']);
   // getMessage(): "The field email is required."
   ```

3. Remember the `+intl-icu` suffix on the domain if your messages use ICU syntax (`{name}`, plural, select). Without it, `symfony/translation` uses plain `%name%`-style substitution instead, and `{name}` is left untouched. See ICU Formatting.

Tips & Tricks

1. Add static factory methods to your exceptions for common cases:

```
class ValidationException extends TranslatableException
{
    public static function required(string $field): self
    {
        return new self(['The field {field} is required.', 'field' => $field]);
    }
}
```

2. Use type hints and PHPDoc for better IDE support:

```
/**
 * @throws ValidationException When the email is invalid.
 */
public function validateEmail(string $email): void
{
    // ...
}
```

Last updated on 09/09/2026