Docs

Translation Project

Derafu Translation

Anonymous · 21/08/2026 · #php

Table of Contents

Translation Project

Derafu Translation

Anonymous · 21/08/2026 · #php

Derafu Translation

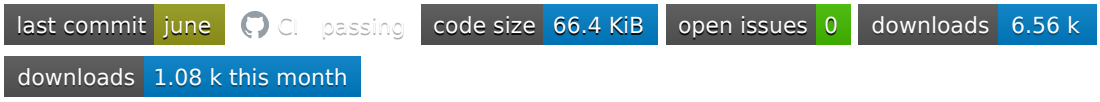
Last updated on 21/08/2026

Introduction

Translation Library with Exception Support

Anonymous · 21/08/2026

Translation Library with Exception Support



A PHP library that solves one specific problem: **making exceptions translatable without compromising your existing code**, while providing powerful ICU message formatting support.

Features

- **Translatable Exceptions**: Make any exception translatable with minimal changes.
- **ICU Support**: Built-in ICU message formatting for complex messages.
- **Multiple Formats**: Load translations from PHP, JSON, YAML files or custom provider.
- **Locale Fallback**: Configurable locale fallback chain.
- **Framework Agnostic**: Works with any system implementing Symfony's TranslatorInterface.
- **Lightweight**: Minimal dependencies (only requires `symfony/translation-contracts`).
- **Extensible**: Easy to add custom message providers.

Installation

```
composer require derafu/translation
```

Basic Usage

Making Exceptions Translatable

```
// Before: Your existing exception.
class ValidationException extends Exception
{
    public function __construct(string $message)
    {
        parent::__construct($message);
    }
}
```

```
// After: Just change the parent class.
class ValidationException extends TranslatableException
{
    // No changes needed!
}
```

Using Translatable Exceptions

```
// 1. Simple string (works like before).
throw new ValidationException('Email is invalid');

// 2. With translation key and parameters.
throw new ValidationException([
    'validation.email.invalid',
    'email' => 'test@example.com'
]);

// 3. With ICU formatting.
throw new ValidationException(
    'The email {email} is not valid',
    ['email' => 'test@example.com']
);

// 4. With complex ICU patterns.
throw new ValidationException(
    '{gender, select, female{She} male{He} other{They}} sent {count, plural, one{#
message} other{# messages}}',
    [
        'gender' => 'female',
        'count' => 5
    ]
);
```

Translation Files

Support for multiple formats:

```
// PHP arrays (fastest).
// translations/messages/en.php
return [
    'welcome' => 'Welcome {name}!',
    'messages' => '{count, plural, =0{No messages} one{# message} other{# messages}}'
];

// JSON.
// translations/messages/en.json
{
    "welcome": "Welcome {name}!",
```

```
    "messages": "{count, plural, =0{No messages} one{# message} other{# messages}}"
}

// YAML (requires symfony/yaml).
# translations/messages/en.yaml
welcome: 'Welcome {name}!'
messages: '{count, plural, =0{No messages} one{# message} other{# messages}}'
```

Setting Up the Translator

```
use Derafu\Translation\Translator;
use Derafu\Translation\Provider\PhpMessageProvider;

// Create provider.
$provider = new PhpMessageProvider(__DIR__ . '/translations');

// Create translator with fallback locales.
$translator = new Translator(
    $provider,
    'en', // Optional. Default `null` for Translator y 'en' for ICU.
    ['es_CL', 'es', 'en'] // Optional. Default are: 'en', 'en_US', 'es', 'es_CL'.
);

// Use in your exception handling.
try {
    // Your code.
} catch (TranslatableException $e) {
    // Get translation for current locale.
    echo $e->trans($translator);

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

Message Providers

The library includes three message providers:

- `PhpMessageProvider`: Uses PHP files returning arrays (fastest).
- `JsonMessageProvider`: Uses JSON files.
- `YamlMessageProvider`: Uses YAML files (requires `symfony/yaml`).

Key Benefits

1. **Zero Compromise**: Keep your existing exception handling code.
2. **ICU Power**: Full ICU message formatting support.

3. **Flexible Loading**: Choose your preferred translation file format.
4. **Clean Separation**: Error logic stays separate from presentation.
5. **Type Safe**: Full PHP 8.3+ type safety.

When to Use This Library

This library is perfect when you:

- Need to internationalize exceptions without refactoring.
- Want ICU message formatting support.
- Need flexible translation file formats.
- Want to keep error handling and translation separate.
- Use exceptions for business logic validation.

Last updated on 21/08/2026

Quick Start

Quick Start Guide

Anonymous · 21/08/2026

Quick Start Guide

Get started with Derafu Translation library in minutes.

Installation

Install via Composer:

```
composer require derafu/translation
```

Optional dependencies:

```
# If you want to use YAML files.
composer require symfony/yaml

# If you want to use Symfony's translator.
composer require symfony/translation
```

Basic Setup

1. Create Translation Files

Choose your preferred format:

```
// translations/messages/en.php
return [
    'welcome' => 'Welcome {name}!',
    'errors' => [
        'required' => 'The field {field} is required.',
        'email' => 'Invalid email: {email}'
    ]
];
```

```
// translations/messages/en.json
{
```

```
    "welcome": "Welcome {name}!",
    "errors.required": "The field {field} is required.",
    "errors.email": "Invalid email: {email}"
}
```

```
# translations/messages/en.yaml
welcome: 'Welcome {name}!'
errors:
  required: 'The field {field} is required.'
  email: 'Invalid email: {email}'
```

2. Create Translator

```
use Derafu\Translation\Translator;
use Derafu\Translation\Provider\PhpMessageProvider;

// Create message provider.
$provider = new PhpMessageProvider(__DIR__ . '/translations');

// Create translator.
$translator = new Translator(
    provider: $provider,
    locale: 'en',
    fallbackLocales: ['en']
);
```

3. Make Exceptions Translatable

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

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

First Translation

Basic Usage

```
// 1. Simple string messages.
throw new ValidationException('Email is required.');

// 2. With translation key.
throw new ValidationException('errors.required', ['field' => 'email']);

// 3. With ICU formatting.
```

```
throw new ValidationException(
    'The value {value} must be between {min} and {max}.',
    [
        'value' => 42,
        'min' => 1,
        'max' => 10,
    ]
);
```

Handling Exceptions

```
try {
    // Your code.
} catch (ValidationException $e) {
    // Get default message.
    echo $e->getMessage();

    // Get translated message.
    echo $e->trans($translator);

    // Get message in specific locale.
    echo $e->trans($translator, 'es');
}
```

Common Use Cases

Form Validation

```
class UserController
{
    public function create(Request $request): Response
    {
        $data = $request->toArray();

        if (empty($data['email'])) {
            throw new ValidationException([
                'errors.required',
                'field' => 'email',
            ]);
        }

        if (!filter_var($data['email'], FILTER_VALIDATE_EMAIL)) {
            throw new ValidationException([
                'errors.email',
                'email' => $data['email'],
            ]);
        }
```

```
        // Create user...
    }
}
```

API Responses

```
class ErrorHandler
{
    public function handle(Throwable $e): JsonResponse
    {
        if ($e instanceof TranslatableException) {
            return new JsonResponse([
                'error' => [
                    'message' => $e->trans($this->translator),
                    'code' => $e->getCode(),
                ]
            ]);
        }

        // Handle other errors...
    }
}
```

Complex Messages

```
// Pluralization.
throw new ValidationException(
    '{count, plural, =0{No files uploaded} one{1 file uploaded} other{# files
uploaded}}',
    ['count' => $fileCount]
);

// Gender.
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 powerful message formatting.
2. Learn about Message Providers for different storage options.
3. Check Real World Examples for common patterns.

4. Explore Advanced Usage for complex scenarios.

Common Pitfalls

1. Always include ‘other’ case in plural/select patterns:

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

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

2. Remember to provide all parameters:

```
// Will fail.
throw new ValidationException(
    'The field {field} is required'
    // Missing parameters!
);

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

3. Use consistent translation keys:

```
// Recommended structure.
validation.required
validation.email.invalid
validation.range.between
```

Tips & Tricks

1. Use constants for translation keys:

```
final class Messages
{
    public const REQUIRED = 'validation.required';
    public const EMAIL_INVALID = 'validation.email.invalid';
}
```

2. Create factory methods for common exceptions:

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

3. Use type hints for better IDE support:

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

Last updated on 21/08/2026

API Reference

API Reference

Anonymous · 21/08/2026

API Reference

Complete reference documentation for the Derafu Translation library.

Interfaces

TranslatableInterface

Extends Symfony's `TranslatorInterface` and adds ICU support.

```
interface TranslatableInterface extends TranslatorInterface, Stringable
{
    /**
     * Converts the message to a string using ICU formatting.
     */
    public function __toString(): string;
}
```

MessageProviderInterface

Defines how translation messages are loaded.

```
interface MessageProviderInterface
{
    /**
     * Returns all messages for a given locale and domain.
     *
     * @param string $locale The locale to load messages for.
     * @param string $domain The translation domain.
     * @return array<string, string> Array of messages where key is the message id.
     */
    public function getMessages(string $locale, string $domain = 'messages'): array;

    /**
     * Returns all available locales for a given domain.
     *
     * @param string $domain The translation domain.
```

```
     * @return array<string> List of available locales.
     */
    public function getAvailableLocales(string $domain = 'messages'): array;
}
```

Classes

TranslatableMessage

Core message class that supports ICU formatting.

```
final class TranslatableMessage implements TranslatableInterface
{
    /**
     * @param string $message The message used for translation.
     * @param array<string, mixed> $parameters Parameters for translation.
     * @param string|null $domain The translation domain.
     * @param string $defaultLocale The locale for ICU formatting.
     */
    public function __construct(
        string $message,
        array $parameters = [],
        ?string $domain = null,
        string $defaultLocale = 'en'
    );

    /**
     * @throws RuntimeException When ICU formatting fails.
     */
    public function __toString(): string;

    public function trans(
        TranslatorInterface $translator,
        ?string $locale = null
    ): string;
}
```

Translator

Main translator implementation, using Symfony's `TranslatorTrait`, with ICU and fallback support.

```
final class Translator implements TranslatorInterface
{
    /**
     * @param MessageProviderInterface $provider Provider of messages.
     * @param string|null $locale Default locale.
     * @param array|null $fallbackLocales Fallback locales chain.
```

```
     */
    public function __construct(
        MessageProviderInterface $provider,
        ?string $locale = null,
        ?array $fallbackLocales = null
    );

    /**
     * @throws IntlException When ICU formatting fails
     */
    public function trans(
        ?string $id,
        array $parameters = [],
        ?string $domain = null,
        ?string $locale = null
    ): string;
}
```

Exceptions

TranslatableException

Base exception class that supports translation.

```
abstract class TranslatableException extends Exception implements
TranslatableInterface
{
    /**
     * @param string|array|TranslatableInterface $message The exception message:
     *   - string: Used as both message and translation key.
     *   - array: First element is message, rest are parameters.
     *   - TranslatableInterface: Used directly.
     * @param int $code The exception code.
     * @param Throwable|null $previous Previous exception.
     * @throws InvalidArgumentException When message array is invalid.
     */
    public function __construct(
        string|array|TranslatableInterface $message,
        int $code = 0,
        ?Throwable $previous = null
    );

    public function trans(
        TranslatorInterface $translator,
        ?string $locale = null
    ): string;
}
```

Message Providers

AbstractMessageProvider

Base class for file-based message providers.

```
abstract class AbstractMessageProvider implements MessageProviderInterface
{
    /**
     * @param string $directory Base directory for translation files.
     * @throws RuntimeException If directory doesn't exist.
     */
    public function __construct(
        protected readonly string $directory
    );

    /**
     * Returns the file extension without dot.
     */
    abstract protected function getFileExtension(): string;

    /**
     * Parses a file into messages array.
     *
     * @throws RuntimeException If file can't be parsed.
     */
    abstract protected function parseFile(string $file): array;
}
```

PhpMessageProvider

Loads translations from PHP files.

```
final class PhpMessageProvider extends AbstractMessageProvider
{
    /**
     * Expected file format:
     * <?php
     * return [
     *     'key' => 'value',
     * ];
     *
     * @throws RuntimeException If file doesn't return array.
     */
    protected function parseFile(string $file): array;
}
```

JsonMessageProvider

Loads translations from JSON files.

```
final class JsonMessageProvider extends AbstractMessageProvider
{
    /**
     * Expected file format:
     * {
     *     "key": "value"
     * }
     *
     * @throws RuntimeException If JSON is invalid.
     */
    protected function parseFile(string $file): array;
}
```

YamlMessageProvider

Loads translations from YAML files (requires symfony/yaml).

```
final class YamlMessageProvider extends AbstractMessageProvider
{
    /**
     * Expected file format:
     * key: value
     *
     * @throws RuntimeException If YAML is invalid.
     */
    protected function parseFile(string $file): array;
}
```

Error Handling

All exceptions extend from base PHP exceptions.

Common exceptions:

- `TranslatableRuntimeException`: General runtime errors.
- `TranslatableInvalidArgumentException`: Invalid input parameters.

Can't extend Derafu Translation exceptions? Then use `TranslatableExceptionTrait` in your exceptions to get the translation capabilities.

Last updated on 21/08/2026

Exceptions

Working with Translatable Exceptions

Anonymous · 21/08/2026

Working with Translatable Exceptions

This guide covers all aspects of working with exceptions in the Derafu Translation library.

Available Exceptions

The library provides translatable versions of all standard PHP exceptions:

Core Exceptions

- `TranslatableException`: Base exception.
- `TranslatableLogicException`: For logical errors.
- `TranslatableRuntimeException`: For runtime errors.

Logic Exceptions

- `TranslatableDomainException`: Domain logic violations.
- `TranslatableInvalidArgumentException`: Invalid input.
- `TranslatableLengthException`: Invalid length.
- `TranslatableOutOfRangeException`: Value out of valid set.

Runtime Exceptions

- `TranslatableOutOfBoundsException`: Invalid index or key.
- `TranslatableOverflowException`: Arithmetic overflow.
- `TranslatableRangeException`: Value not within range.
- `TranslatableUnderflowException`: Arithmetic underflow.
- `TranslatableUnexpectedValueException`: Unexpected value type.

Using the Trait

TranslatableExceptionTrait

If you want to add translation capabilities to your own exceptions, you can use the trait:

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

class MyCustomException extends DomainException implements TranslatableInterface
{
    use TranslatableExceptionTrait;
}
```

When to Use the Trait vs Extending Base Exceptions

Use the trait when:

- You already extend another exception.
- You need to customize the exception behavior.
- You want to add additional functionality.

Use the base exceptions when:

- You don’t need custom behavior.
- You want the simplest implementation.
- You’re creating domain-specific exceptions.

Example with trait:

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

    protected string $defaultDomain = 'orders';

    public static function insufficientStock(Product $product): self
    {
        return new self([
            'order.insufficient_stock',
            'product' => $product->getName(),
        ]);
    }
}
```

Example with inheritance:

```
class ValidationException extends TranslatableDomainException
{
    // Inherits all translation functionality.
}
```

When to Use Each Exception

TranslatableException

Base exception, use when:

- General error conditions.
- No specific exception type fits.
- Creating custom exception hierarchies.

Real-world examples:

1. Generic API errors.

```
throw new TranslatableException([
    'api.error.generic',
    'code' => $errorCode,
]);
```

2. System configuration errors.

```
throw new TranslatableException([
    'system.config.missing',
    'key' => 'database.host',
]);
```

3. Third-party service errors.

```
throw new TranslatableException([
    'service.unavailable',
    'service' => 'payment',
    'reason' => $response->getError(),
]);
```

TranslatableLogicException

For programming errors that can be detected during development:

1. Invalid business rule implementation.

```
throw new TranslatableLogicException([
    'logic.invalid_workflow',
    'state' => $currentState,
    'action' => $attemptedAction,
]);
```

2. Configuration errors.

```
throw new TranslatableLogicException([
    'config.invalid_combination',
    'option1' => $value1,
    'option2' => $value2,
]);
```

3. Invalid method usage.

```
throw new TranslatableLogicException([
    'method.invalid_order',
    'method' => 'process',
    'required_before' => 'validate',
]);
```

TranslatableDomainException

For violations of domain rules:

1. Business rule violations.

```
throw new TranslatableDomainException([
    'order.invalid_status_transition',
    'from' => $currentStatus,
    'to' => $newStatus,
]);
```

2. Invalid entity state.

```
throw new TranslatableDomainException([
    'product.cannot_publish',
    'reason' => 'missing_price',
]);
```

3. Business constraint violations.

```
throw new TranslatableDomainException([
    'account.overdraft_limit_exceeded',
    'amount' => $amount,
    'limit' => $overdraftLimit,
]);
```

TranslatableInvalidArgumentException

For invalid input values:

1. Invalid parameter types.

```
throw new TranslatableInvalidArgumentException([
    'argument.invalid_type',
    'argument' => 'date',
    'expected' => 'DateTime',
    'received' => get_debug_type($date),
]);
```

2. Invalid format.

```
throw new TranslatableInvalidArgumentException([
    'argument.invalid_format',
    'field' => 'phone',
    'format' => '+XX-XXX-XXXXXX',
]);
```

3. Invalid options.

```
throw new TranslatableInvalidArgumentException([
    'argument.invalid_option',
    'option' => 'sort',
    'valid' => implode(', ', $validOptions),
]);
```

TranslatableLengthException

For invalid lengths:

1. String length violations.

```
throw new TranslatableLengthException([
```

```
    'length.string_too_long',
    'field' => 'title',
    'max' => 255,
    'current' => strlen($title),
]);
```

2. Collection size issues.

```
throw new TranslatableLengthException([
    'length.too_many_items',
    'max' => 10,
    'current' => count($items),
]);
```

3. Buffer size problems.

```
throw new TranslatableLengthException([
    'length.buffer_overflow',
    'size' => $bufferSize,
    'required' => $requiredSize,
]);
```

TranslatableOutOfRangeException

For values outside valid set:

1. Invalid enum values.

```
throw new TranslatableOutOfRangeException([
    'value.invalid_status',
    'value' => $status,
    'valid' => implode(', ', Status::cases()),
]);
```

2. Invalid date ranges.

```
throw new TranslatableOutOfRangeException([
    'date.out_of_range',
    'date' => $date->format('Y-m-d'),
    'min' => $minDate->format('Y-m-d'),
    'max' => $maxDate->format('Y-m-d'),
]);
```

3. Invalid numerical ranges.

```
throw new TranslatableOutOfRangeException([
    'value.out_of_range',
    'value' => $percentage,
    'min' => 0,
    'max' => 100,
]);
```

TranslatableOutOfBoundsException

For invalid array/collection access:

1. Invalid array index.

```
throw new TranslatableOutOfBoundsException([
    'index.invalid',
    'index' => $index,
    'max' => count($array) - 1,
]);
```

2. Invalid page number.

```
throw new TranslatableOutOfBoundsException([
    'page.invalid',
    'page' => $page,
    'total' => $totalPages,
]);
```

3. Invalid collection access.

```
throw new TranslatableOutOfBoundsException([
    'record.not_found',
    'id' => $id,
    'type' => 'user',
]);
```

TranslatableRangeException

For values technically valid but not allowed:

1. Value scale errors.

```
throw new TranslatableRangeException([
    'number.too_many_decimals',
    'value' => $number,
    'max_decimals' => 2,
]);
```

2. Business range violations.

```
throw new TranslatableRangeException([
    'discount.exceeds_limit',
    'discount' => $discount,
    'max_allowed' => $maxDiscount,
]);
```

3. Technical limitations.

```
throw new TranslatableRangeException([
    'file.too_large',
    'size' => $fileSize,
    'max' => $maxUploadSize,
]);
```

TranslatableUnexpectedValueException

For values of unexpected type:

1. Invalid return values.

```
throw new TranslatableUnexpectedValueException([
    'api.unexpected_response',
    'expected' => 'array',
    'received' => gettype($response),
]);
```

2. Invalid data format.

```
throw new TranslatableUnexpectedValueException([
    'data.invalid_format',
    'expected' => 'JSON',
    'received' => $detectedFormat,
]);
```

3. Invalid state values.

```
throw new TranslatableUnexpectedValueException([
    'state.unexpected',
    'state' => $currentState,
    'expected' => implode(', ', $validStates),
]);
```

Best Practices

1. **Choose Specific Exceptions**
    - Use the most specific exception type available.
    - Consider the error's nature (logic vs runtime).
    - Think about error recovery possibilities.
2. **Consistent Domain Language**
    - Use consistent translation keys.
    - Group related messages.
    - Use clear, descriptive keys.
3. **Provide Context**
    - Include relevant parameters.
    - Add debugging information.
    - Consider logging needs.
4. **Exception Hierarchy**
    - Create domain-specific exceptions.
    - Extend appropriate base classes.
    - Use meaningful inheritance chains.
5. **Translation Keys**
    - Use consistent naming patterns.
    - Group by domain/module.
    - Include error type in key.

Example hierarchy

```
// Base exception for your domain.
abstract class OrderException extends TranslatableDomainException
{
```

```
    protected string $defaultDomain = 'orders';
}

// Specific exceptions.
class OrderNotFoundException extends OrderException {}
class OrderValidationException extends OrderException {}
class OrderStateException extends OrderException {}
```

Last updated on 21/08/2026

ICU Formatting

ICU Message Formatting Guide

Anonymous · 21/08/2026

ICU Message Formatting Guide

This guide covers ICU (International Components for Unicode) message formatting in the context of the Derafu Translation library. ICU provides powerful tools for handling pluralization, gender, and complex message patterns.

Basic Placeholders

The simplest form of ICU formatting uses named placeholders:

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

Multiple placeholders are supported:

```
throw new ValidationException(
    'Value {value} for field {field} is invalid.',
    [
        'value' => 'test@',
        'field' => 'email',
    ],
);
// Output: "Value test@ for field email is invalid."
```

Pluralization

ICU provides robust pluralization support with multiple forms:

```
// Basic pluralization.
$message = '{count, plural, =0{No messages} one{# message} other{# messages}}';
$translator->trans($message, ['count' => 2]);  // "2 messages"
$translator->trans($message, ['count' => 1]);  // "1 message"
$translator->trans($message, ['count' => 0]);  // "No messages"
```

```
// With exact matches and ranges.
$message = '{count, plural,
    =0 {No items}
    =1 {One item}
    =2 {A couple of items}
    other {# items}
}';
```

Available plural categories:

- `zero`: For languages with special zero forms.
- `one`: Singular form.
- `two`: Dual form (for languages that have it).
- `few`: For languages with special handling of small numbers.
- `many`: For languages with special handling of large numbers.
- `other`: Default form (required).
- `=n`: Exact number matches.

Gender Selection

Gender-based message formatting:

```
$message = '{gender, select,
    female {She liked your post}
    male {He liked your post}
    other {They liked your post}
}';

$translator->trans($message, ['gender' => 'female']);
// Output: "She liked your post".
```

Complex gender example with variables:

```
$message = '{gender, select,
    female {{name} added her comment}
    male {{name} added his comment}
    other {{name} added their comment}
}';

$translator->trans($message, [
    'gender' => 'female',
    'name' => 'Alice'
]);
// Output: "Alice added her comment".
```

Number Formatting

ICU supports various number formats:

```
// Basic number.
'{value, number}'

// With minimum decimals.
'{value, number, .00}'

// Percentage.
'{value, number, percent}'

// Currency.
'{value, number, currency}'
```

Example in context:

```
throw new ValidationException(
    'Balance must be greater than {min, number, currency}.',
    ['min' => 100],
);
// Output: "Balance must be greater than $100.00".
```

Nested Formatting

ICU patterns can be nested for complex scenarios:

```
$message = '{gender, select,
    female {
        {count, plural,
            =0 {She has no messages}
            one {She has # message}
            other {She has # messages}
        }
    }
    male {
        {count, plural,
            =0 {He has no messages}
            one {He has # message}
            other {He has # messages}
        }
    }
    other {
        {count, plural,
            =0 {They have no messages}
```

```
            one {They have # message}
            other {They have # messages}
        }
    }
}';

$translator->trans($message, [
    'gender' => 'female',
    'count' => 5,
]);
// Output: "She has 5 messages".
```

Common Patterns

Here are some common patterns used in validation messages:

```
// Range validation.
'Value must be between {min} and {max}.'

// List validation.
'{count, plural,
    =0 {List cannot be empty}
    one {At least one item is required}
    other {At least # items are required}
}'

// Status messages.
'{status, select,
    pending {Waiting for approval}
    approved {Approved on {date}}
    rejected {Rejected: {reason}}
    other {Unknown status}
}'

// File validation.
'{type, select,
    image {Only images are allowed}
    document {Only documents are allowed}
    other {Invalid file type}
}, max size: {size}'
```

Troubleshooting

Common issues and solutions:

1. **Missing ‘other’ category**

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

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

2. **Invalid nesting**

```
// Wrong.
'{outer, select, a{{inner}}'

// Correct.
'{outer, select, a {{inner}} other {default}}'
```

3. **Unmatched braces**

```
// Wrong.
'Hello {name'

// Correct.
'Hello {name}'
```

4. **Missing parameters**

```
// Will fail if 'count' is not provided.
'{count} items'
```

Remember:

- Always include the ‘other’ case in select/plural patterns.
- Check braces are properly matched.
- Ensure all used parameters are provided.
- Test with different values and locales.

For more information about ICU message format:

- ICU User Guide
- MessageFormat Guide

Last updated on 21/08/2026

Custom Providers

Creating Custom Message Providers

Anonymous · 21/08/2026

Creating Custom Message Providers

This guide explains how to create custom message providers for different translation storage formats and sources.

Provider Interface

All message providers must implement the `MessageProviderInterface`:

```
interface MessageProviderInterface
{
    /**
     * Returns all messages for a given locale and domain.
     *
     * @param string $locale The locale to load messages for.
     * @param string $domain The translation domain.
     * @return array<string, string> Array of messages where key is the message id.
     */
    public function getMessages(string $locale, string $domain = 'messages'): array;

    /**
     * Returns all available locales for a given domain.
     *
     * @param string $domain The translation domain.
     * @return array<string> List of available locales.
     */
    public function getAvailableLocales(string $domain = 'messages'): array;
}
```

Basic Implementation

Here’s a simple example of a custom provider that loads messages from a database:

```
use Derafu\Translation\Contract\MessageProviderInterface;
use PDO;

final class DatabaseMessageProvider implements MessageProviderInterface
```

```
{
    public function __construct(
        private readonly PDO $db,
        private readonly string $table = 'translations'
    ) {
    }

    public function getMessages(string $locale, string $domain = 'messages'): array
    {
        $stmt = $this->db->prepare(
            "SELECT message_key, message_text
             FROM {$this->table}
             WHERE locale = ? AND domain = ?"
        );
        $stmt->execute([$locale, $domain]);

        return $stmt->fetchAll(PDO::FETCH_KEY_PAIR);
    }

    public function getAvailableLocales(string $domain = 'messages'): array
    {
        $stmt = $this->db->prepare(
            "SELECT DISTINCT locale
             FROM {$this->table}
             WHERE domain = ?"
        );
        $stmt->execute([$domain]);

        return $stmt->fetchAll(PDO::FETCH_COLUMN);
    }
}
```

Using the Abstract Provider

For file-based providers, you can extend the `AbstractMessageProvider`:

```
use Derafu\Translation\Abstract\AbstractMessageProvider;

final class IniMessageProvider extends AbstractMessageProvider
{
    protected function getFileExtension(): string
    {
        return 'ini';
    }

    protected function parseFile(string $file): array
    {
        $messages = parse_ini_file($file, false);
```

```
        if ($messages === false) {
            throw new RuntimeException(
                sprintf('Could not parse INI file "%s".', $file)
            );
        }

        return $messages;
    }
}
```

The abstract provider handles:

- Directory structure validation.
- File path generation.
- Available locales discovery.

You only need to implement:

- `getFileExtension()`: Returns the file extension.
- `parseFile()`: Parses the file content into messages array.

Other Examples

Redis Provider

```
use Redis;
use Derafu\Translation\Contract\MessageProviderInterface;

final class RedisMessageProvider implements MessageProviderInterface
{
    public function __construct(
        private readonly Redis $redis,
        private readonly string $prefix = 'translations:'
    ) {
    }

    public function getMessages(string $locale, string $domain = 'messages'): array
    {
        $key = "{$this->prefix}{$domain}:{$locale}";
        $messages = $this->redis->hGetAll($key);

        return $messages ?: [];
    }

    public function getAvailableLocales(string $domain = 'messages'): array
    {
        $pattern = "{$this->prefix}{$domain}:*";
        $keys = $this->redis->keys($pattern);
```

```
        return array_map(
            fn($key) => substr($key, strrpos($key, ':') + 1),
            $keys
        );
    }
}
```

API Provider

```
use GuzzleHttp\Client;
use Derafu\Translation\Contract\MessageProviderInterface;

final class ApiMessageProvider implements MessageProviderInterface
{
    public function __construct(
        private readonly Client $client,
        private readonly string $baseUrl
    ) {
    }

    public function getMessages(string $locale, string $domain = 'messages'): array
    {
        $response = $this->client->get(
            "{$this->baseUrl}/translations/{$domain}/{$locale}"
        );

        return json_decode(
            $response->getBody()->getContents(),
            true
        );
    }

    public function getAvailableLocales(string $domain = 'messages'): array
    {
        $response = $this->client->get(
            "{$this->baseUrl}/translations/{$domain}/locales"
        );

        return json_decode(
            $response->getBody()->getContents(),
            true
        );
    }
}
```

Best Practices

1. **Error Handling**

```
protected function parseFile(string $file): array
{
    try {
        // Parse file.
    } catch (Exception $e) {
        throw new RuntimeException(
            sprintf('Error parsing file "%s": %s', $file, $e->getMessage())
        );
    }
}
```

2. **Caching Support**

```
final class CachedProvider implements MessageProviderInterface
{
    public function __construct(
        private readonly MessageProviderInterface $provider,
        private readonly CacheInterface $cache,
        private readonly int $ttl = 3600
    ) {
    }

    public function getMessages(string $locale, string $domain = 'messages'):
array
    {
        $key = "translations:{$domain}:{$locale}";

        return $this->cache->remember($key, $this->ttl, function() use ($locale,
$domain) {
            return $this->provider->getMessages($locale, $domain);
        });
    }
}
```

3. **Validation**

```
private function validateMessages(array $messages): void
{
    foreach ($messages as $key => $value) {
        if (!is_string($key)) {
            throw new RuntimeException('Message keys must be strings.');
        }
        if (!is_string($value)) {
            throw new RuntimeException('Message values must be strings.');
        }
```

```
    }
}
```

4. **Logging and Debugging**

```
public function getMessages(string $locale, string $domain = 'messages'): array
{
    $messages = $this->loadMessages($locale, $domain);

    if (empty($messages)) {
        $this->logger->warning(
            'No messages found for locale {locale} and domain {domain}.',
            ['locale' => $locale, 'domain' => $domain]
        );
    }

    return $messages;
}
```

Remember:

- Always validate input and output.
- Handle errors gracefully.
- Consider implementing caching for performance.
- Add logging for debugging.
- Keep providers focused and single-purpose.
- Use dependency injection for external services.

Last updated on 21/08/2026

Advanced Usage

Advanced Usage Guide

Anonymous · 21/08/2026

Advanced Usage Guide

This guide covers advanced patterns and usage scenarios for the Derafu Translation library.

Message Composition

Base Messages

```
class MessageTemplates
{
    public const REQUIRED = 'validation.required';
    public const INVALID = 'validation.invalid';
    public const FORMAT = 'validation.format';

    public static function required(string $field): TranslatableInterface
    {
        return new TranslatableMessage(self::REQUIRED, ['field' => $field]);
    }

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

    public static function format(string $field, string $format):
TranslatableInterface
    {
        return new TranslatableMessage(self::FORMAT, [
            'field' => $field,
            'format' => $format
        ]);
    }
}
```

Composite Messages

```
class CompositeMessage implements TranslatableInterface
{
    /** @var TranslatableInterface[] */
    private array $messages;

    /**
     * @param TranslatableInterface[] $messages
     */
    public function __construct(array $messages)
    {
        $this->messages = $messages;
    }

    public function trans(TranslatorInterface $translator, ?string $locale = null):
string
    {
        return implode(' ', array_map(
            fn(TranslatableInterface $message) => $message->trans($translator,
$locale),
            $this->messages
        ));
    }

    public function __toString(): string
    {
        return implode(' ', array_map(
            fn(TranslatableInterface $message) => (string) $message,
            $this->messages
        ));
    }
}

// Usage
throw new ValidationException(new CompositeMessage([
    MessageTemplates::required('email'),
    MessageTemplates::format('email', 'user@example.com')
]));
```

Dynamic Translation Loading

Translation Strategy

```
interface TranslationStrategy
{
    public function getMessages(string $locale): array;
    public function supports(string $domain): bool;
```

```
}

class ApiTranslationStrategy implements TranslationStrategy
{
    public function __construct(
        private readonly HttpClientInterface $client,
        private readonly string $apiUrl
    ) {
    }

    public function getMessages(string $locale): array
    {
        $response = $this->client->request('GET', sprintf(
            '%s/translations/%s',
            $this->apiUrl,
            $locale
        ));

        return json_decode($response->getBody()->getContents(), true);
    }

    public function supports(string $domain): bool
    {
        return $domain === 'remote';
    }
}

class DynamicProvider implements MessageProviderInterface
{
    /** @var TranslationStrategy[] */
    private array $strategies = [];

    public function addStrategy(TranslationStrategy $strategy): void
    {
        $this->strategies[] = $strategy;
    }

    public function getMessages(string $locale, string $domain = 'messages'): array
    {
        foreach ($this->strategies as $strategy) {
            if ($strategy->supports($domain)) {
                return $strategy->getMessages($locale);
            }
        }
        return [];
    }
}
```

Exception Hierarchies

Base Domain Exception

```
abstract class DomainException extends TranslatableException
{
    protected function getDefaultDomain(): string
    {
        return 'domain';
    }

    protected function getMessageContext(): array
    {
        return [
            'timestamp' => date('Y-m-d H:i:s'),
            'trace_id' => $this->getTraceId()
        ];
    }

    private function getTraceId(): string
    {
        return bin2hex(random_bytes(16));
    }
}
```

Specific Exceptions

```
class OrderException extends DomainException
{
    protected function getDefaultDomain(): string
    {
        return 'orders';
    }

    public static function insufficientStock(
        Product $product,
        int $requested,
        int $available
    ): self {
        return new self([
            'order.insufficient_stock',
            'product' => $product->getName(),
            'requested' => $requested,
            'available' => $available
        ]);
    }
}
```

```
class PaymentException extends DomainException
{
    protected function getDefaultDomain(): string
    {
        return 'payments';
    }

    public static function insufficientFunds(
        Money $amount,
        Money $balance
    ): self {
        return new self([
            'payment.insufficient_funds',
            'amount' => $amount->format(),
            'balance' => $balance->format(),
            'currency' => $amount->getCurrency()
        ]);
    }
}
```

Context-Aware Translation

Translation Context

```
class TranslationContext
{
    public function __construct(
        private readonly array $data = []
    ) {
    }

    public function get(string $key, mixed $default = null): mixed
    {
        return $this->data[$key] ?? $default;
    }

    public function merge(array $data): self
    {
        return new self(array_merge($this->data, $data));
    }
}

class ContextAwareTranslator implements TranslatorInterface
{
    private ?TranslationContext $context = null;

    public function __construct(
        private readonly TranslatorInterface $translator
```

```
    ) {
    }

    public function withContext(TranslationContext $context): self
    {
        $clone = clone $this;
        $clone->context = $context;
        return $clone;
    }

    public function trans(
        ?string $id,
        array $parameters = [],
        ?string $domain = null,
        ?string $locale = null
    ): string {
        if ($this->context) {
            $parameters = array_merge(
                $parameters,
                $this->context->toArray()
            );
        }

        return $this->translator->trans($id, $parameters, $domain, $locale);
    }
}
```

Translation Decorators

Logging Decorator

```
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,
            'parameters' => $parameters,
            'domain' => $domain,
            'locale' => $locale,
            'result' => $result
        ]);

        return $result;
    }
}
```

Fallback Chain Decorator

```
class FallbackChainTranslator implements TranslatorInterface
{
    /** @var TranslatorInterface[] */
    private array $translators;

    public function __construct(TranslatorInterface ...$translators)
    {
        $this->translators = $translators;
    }

    public function trans(
        ?string $id,
        array $parameters = [],
        ?string $domain = null,
        ?string $locale = null
    ): string {
        $lastException = null;

        foreach ($this->translators as $translator) {
            try {
                return $translator->trans($id, $parameters, $domain, $locale);
            } catch (Exception $e) {
                $lastException = $e;
                continue;
            }
        }

        throw new RuntimeException(
            'No translator could handle the translation',
            0,
            $lastException
        );
    }
}
```

Caching Decorator

```
class CachingTranslator implements TranslatorInterface
{
    public function __construct(
        private readonly TranslatorInterface $translator,
        private readonly CacheInterface $cache,
        private readonly int $ttl = 3600
    ) {
    }

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

        return $this->cache->get($key, function() use ($id, $parameters, $domain,
$locale) {
            return $this->translator->trans($id, $parameters, $domain, $locale);
        }, $this->ttl);
    }

    private function getCacheKey(
        string $id,
        array $parameters,
        ?string $domain,
        ?string $locale
    ): string {
        return md5(serialize([
            'id' => $id,
            'parameters' => $parameters,
            'domain' => $domain,
            'locale' => $locale
        ]));
    }
}
```

Remember:

- Use composition to extend functionality.
- Keep single responsibility principle.
- Make exceptions domain-specific.
- Consider performance implications.
- Add proper logging and monitoring.

- Handle edge cases gracefully.

Last updated on 21/08/2026

Real World

Real World Examples

Anonymous · 21/08/2026

Real World Examples

This guide provides practical examples of using the Derafu Translation library in real-world scenarios.

REST API Error Handling

Error Response Structure

```
class ApiException extends TranslatableException
{
    public function toArray(): array
    {
        return [
            'error' => [
                'message' => $this->getMessage(),
                'code' => $this->getCode(),
                'type' => $this->getType(),
            ],
        ];
    }

    protected function getType(): string
    {
        return (new ReflectionClass($this))->getShortName();
    }
}

class ValidationApiException extends ApiException
{
    private array $errors;

    public function __construct(array $errors, int $code = 422)
    {
        $this->errors = $errors;
        parent::__construct('validation.failed', $code);
    }

    public function toArray(): array
    {
```

```
        return [
            'error' => [
                'message' => $this->getMessage(),
                'code' => $this->getCode(),
                'type' => $this->getType(),
                'errors' => $this->errors,
            ],
        ];
    }
}
```

API Error Handler

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

    public function handle(Throwable $error): JsonResponse
    {
        if ($error instanceof ApiException) {
            $data = $error->toArray();
            if ($error instanceof TranslatableException) {
                $data['error']['message'] = $error->trans(
                    $this->translator,
                    $this->getLocaleFromRequest()
                );
            }
            return new JsonResponse($data, $error->getCode());
        }

        // Handle other errors...
    }
}
```

Usage in Controllers

```
class UserController
{
    public function create(Request $request): JsonResponse
    {
        $data = $request->toArray();

        if (empty($data['email'])) {
            throw new ValidationApiException([
                'email' => new TranslatableMessage(
                    'validation.required',
```

```
                    ['field' => 'email']
                ),
            ]);
        }

        try {
            // Create user...
        } catch (DuplicateEmailException $e) {
            throw new ValidationApiException([
                'email' => new TranslatableMessage(
                    'validation.email.duplicate',
                    ['email' => $data['email']]
                ),
            ]);
        }
    }
}
```

Form Validation

Form Type

```
class RegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('email', EmailType::class, [
                'constraints' => [
                    new NotBlank([
                        'message' => new TranslatableMessage(
                            'validation.required',
                            ['field' => 'email']
                        ),
                    ]),
                    new Email([
                        'message' => new TranslatableMessage(
                            'validation.email.invalid',
                            ['email' => '{{ value }}']
                        ),
                    ])
                ]
            ])
            ->add('password', PasswordType::class, [
                'constraints' => [
                    new Length([
                        'min' => 8,
                        'minMessage' => new TranslatableMessage(
```

```
                            'validation.password.min_length',
                            ['min' => 8]
                        ),
                    ])
                ]
            ]);
    }
}
```

Form Handler

```
class RegistrationFormHandler
{
    public function handle(FormInterface $form): User
    {
        if (!$form->isSubmitted()) {
            throw new FormException('form.not_submitted');
        }

        if (!$form->isValid()) {
            $errors = [];
            foreach ($form->getErrors(true) as $error) {
                $errors[$error->getOrigin()->getName()] = $error->getMessage();
            }
            throw new ValidationApiException($errors);
        }

        return $this->createUser($form->getData());
    }
}
```

Domain-Specific Validation

Order Processing

```
class OrderProcessor
{
    public function process(Order $order): void
    {
        // Check stock.
        if (!$this->hasStock($order)) {
            throw new OrderException([
                'order.insufficient_stock',
                'product' => $order->getProduct()->getName(),
                'requested' => $order->getQuantity(),
                'available' => $this->getAvailableStock($order->getProduct()),
            ]);
```

```
        }

        // Check status transitions.
        if (!$this->canTransition($order, $status)) {
            throw new OrderException([
                'order.invalid_transition',
                'from' => $order->getStatus(),
                'to' => $status,
                'allowed' => implode(', ', $this->getAllowedTransitions($order)),
            ]);
        }
    }
}
```

Financial Validation

```
class PaymentValidator
{
    public function validate(Payment $payment): void
    {
        // Balance check.
        if ($payment->getAmount() > $this->getBalance()) {
            throw new PaymentException([
                'payment.insufficient_funds',
                'amount' => $payment->getAmount(),
                'balance' => $this->getBalance(),
                'currency' => $payment->getCurrency(),
            ]);
        }

        // Limit check.
        if ($payment->getAmount() > $this->getDailyLimit()) {
            throw new PaymentException([
                'payment.limit_exceeded',
                'amount' => $payment->getAmount(),
                'limit' => $this->getDailyLimit(),
                'period' => 'daily',
            ]);
        }
    }
}
```

Complex Business Rules

Document Workflow

```
class DocumentWorkflow
```

```
{
    public function validate(Document $document): void
    {
        // Check permissions.
        if (!$this->canUserAccess($document)) {
            throw new DocumentException([
                'document.access_denied',
                'document' => $document->getTitle(),
                'user' => $this->getCurrentUser()->getName(),
                'required_role' => $document->getRequiredRole(),
            ]);
        }

        // Check workflow state.
        if (!$this->canTransition($document, $action)) {
            throw new DocumentException([
                'document.invalid_workflow_transition',
                'document' => $document->getTitle(),
                'current' => $document->getState(),
                'action' => $action,
                'required_approvals' => $document->getRequiredApprovals(),
                'current_approvals' => $document->getCurrentApprovals(),
            ]);
        }
    }
}
```

Multiple Translation Sources

Combining Providers

```
class CompositeProvider implements MessageProviderInterface
{
    /** @var MessageProviderInterface[] */
    private array $providers;

    public function __construct(array $providers)
    {
        $this->providers = $providers;
    }

    public function getMessages(string $locale, string $domain = 'messages'): array
    {
        $messages = [];
        foreach ($this->providers as $provider) {
            $messages = array_merge(
                $messages,
                $provider->getMessages($locale, $domain)
```

```
            );
        }
        return $messages;
    }
}

// Usage.
$provider = new CompositeProvider([
    new PhpMessageProvider(__DIR__ . '/translations'),
    new DatabaseMessageProvider($db),
    new RedisMessageProvider($redis),
]);
```

Cache Layer

```
class CachedProvider implements MessageProviderInterface
{
    public function __construct(
        private readonly MessageProviderInterface $provider,
        private readonly CacheInterface $cache,
        private readonly int $ttl = 3600
    ) {
    }

    public function getMessages(string $locale, string $domain = 'messages'): array
    {
        $key = "translations:{$domain}:{$locale}";

        return $this->cache->get($key, function() use ($locale, $domain) {
            return $this->provider->getMessages($locale, $domain);
        }, $this->ttl);
    }
}
```

Remember:

- Keep error messages user-friendly but informative.
- Include relevant context in error messages.
- Use consistent message structure.
- Consider performance with caching.
- Handle nested validations properly.
- Plan for internationalization from the start.

Last updated on 21/08/2026

Symfony Integration

Symfony Integration Guide

Anonymous · 21/08/2026

Symfony Integration Guide

This guide explains how to integrate the Derafu Translation library with Symfony's translation system.

Basic Integration

The simplest way to use Derafu Translation with Symfony is to use Symfony's translator directly:

```
use Symfony\Component\Translation\Translator;
use Symfony\Component\Translation\Loader\YamlFileLoader;

class ErrorHandler
{
    public function __construct(
        private readonly Translator $translator
    ) {
    }

    public function handle(TranslatableException $e): void
    {
        $message = $e->trans($this->translator);
        // Handle translated message.
    }
}
```

Using Symfony's Translator

Configuration

```
# config/packages/translation.yaml
framework:
    default_locale: 'en'
    translator:
        default_path: '%kernel.project_dir%/translations'
        fallbacks:
            - 'en'
        paths:
```

```
            - '%kernel.project_dir%/vendor/your-vendor/your-package/translations'
```

Translation Files

```
# translations/errors.en.yaml
validation:
    required: 'The field {field} is required'
    email:
        invalid: 'The email {email} is not valid'
    min_length: 'The field {field} must be at least {min} characters'

# translations/errors.es.yaml
validation:
    required: 'El campo {field} es requerido'
    email:
        invalid: 'El email {email} no es válido'
    min_length: 'El campo {field} debe tener al menos {min} caracteres'
```

Exception Handler

```
namespace App\EventListener;

use Symfony\Component\HttpKernel\Event\ExceptionEvent;
use Symfony\Contracts\Translation\TranslatorInterface;
use Derafu\Translation\Exception\Core\TranslatableException;

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

    public function onKernelException(ExceptionEvent $event): void
    {
        $throwable = $event->getThrowable();

        if (!$throwable instanceof TranslatableException) {
            return;
        }

        $response = new JsonResponse([
            'error' => $throwable->trans(
                $this->translator,
                $this->translator->getLocale()
            ),
        ]);

        $event->setResponse($response);
```

```
    }
}
```

Service Configuration

```
# config/services.yaml
services:
    App\EventListener\TranslatableExceptionListener:
        tags:
            - { name: kernel.event_listener, event: kernel.exception }
```

Advanced Usage

Custom Exception Response Format

```
class ApiExceptionListener
{
    public function onKernelException(ExceptionEvent $event): void
    {
        $throwable = $event->getThrowable();

        if (!$throwable instanceof TranslatableException) {
            return;
        }

        $response = new JsonResponse([
            'status' => 'error',
            'message' => [
                'text' => $throwable->trans($this->translator),
                'translation_key' => $throwable->getTranslatableMessage()->getId(),
                'parameters' => $throwable->getTranslatableMessage()->getParameters(),
            ],
            'code' => $throwable->getCode(),
        ]);

        $event->setResponse($response);
    }
}
```

Locale Based on User Preferences

```
class LocaleListener
{
    public function onKernelRequest(RequestEvent $event): void
    {
        $request = $event->getRequest();
```

```
        // Get locale from user preferences.
        $locale = $request->getPreferredLanguage(['en', 'es']);

        // Set for current request.
        $request->setLocale($locale);

        // Set for translator.
        $this->translator->setLocale($locale);
    }
}
```

Best Practices

1. **Organize Translation Files**

```
translations/
├── errors/
│   ├── validators.en.yaml
│   ├── validators.es.yaml
│   ├── forms.en.yaml
│   └── forms.es.yaml
└── messages/
    ├── app.en.yaml
    └── app.es.yaml
```

2. **Use Constants for Translation Keys**

```
final class TranslationKeys
{
    public const VALIDATION_REQUIRED = 'validation.required';
    public const VALIDATION_EMAIL = 'validation.email.invalid';
    // ...
}
```

3. **Create Exception Factory**

```
final class ValidationExceptionFactory
{
    public static function required(string $field): ValidationException
    {
        return new ValidationException([
            TranslationKeys::VALIDATION_REQUIRED,
            'field' => $field
        ]);
    }
```

```
}
```

4. **Log Missing Translations**

```
$this->translator->setFallbackLocales(['en']);
$this->translator->addListener(
    TranslationEvents::MISSING_TRANSLATION,
    function(MissingTranslationEvent $event) {
        $this->logger->warning(
            'Missing translation: {key} for locale {locale}',
            [
                'key' => $event->getMessageId(),
                'locale' => $event->getLocale()
            ]
        );
    }
);
```

Remember:

- Keep translation files organized by domain.
- Use constants for translation keys to avoid typos.
- Create factories for common exceptions.
- Log missing translations in development.
- Use fallback locales appropriately.
- Consider user preferences for locale selection.

Last updated on 21/08/2026

Testing

Testing Guide

Anonymous · 21/08/2026

Testing Guide

This guide covers how to test applications using the Derafu Translation library, focusing on testing translatable exceptions and message handling.

Testing Exceptions

Basic Exception Testing

```
use Derafu\Translation\TranslatableMessage;
use PHPUnit\Framework\TestCase;

class ValidationExceptionTest extends TestCase
{
    public function testBasicException(): void
    {
        $exception = new ValidationException('Email is invalid.');

        // Without translation, should use original message.
        $this->assertEquals('Email is invalid.', $exception->getMessage());
    }

    public function testExceptionWithParameters(): void
    {
        $exception = new ValidationException([
            'validation.email',
            'email' => 'test@example',
        ]);

        // Test ICU formatting without translator.
        $this->assertEquals(
            'Invalid email: test@example',
            $exception->getMessage()
        );
    }

    public function testWithTranslatableMessage(): void
    {
        $message = new TranslatableMessage(
```

```
            'validation.required',
            ['field' => 'email'],
        );

        $exception = new ValidationException($message);
        $this->assertInstanceOf(
            TranslatableMessage::class,
            $exception->getTranslatableMessage()
        );
    }
}
```

Using Mock Translator

```
class OrderExceptionTest extends TestCase
{
    private MockObject&TranslatorInterface $translator;

    protected function setUp(): void
    {
        $this->translator = $this->createMock(TranslatorInterface::class);
    }

    public function testOrderValidation(): void
    {
        $this->translator
            ->expects($this->once())
            ->method('trans')
            ->with(
                'order.invalid_status',
                ['status' => 'pending'],
                'errors',
                'en'
            )
            ->willReturn('Invalid order status: pending');

        $exception = new OrderException([
            'order.invalid_status',
            'status' => 'pending'
        ]);

        $this->assertEquals(
            'Invalid order status: pending',
            $exception->trans($this->translator, 'en')
        );
    }
}
```

Testing Translations

Testing Message Files

```
class TranslationFilesTest extends TestCase
{
    private string $fixturesDir;

    protected function setUp(): void
    {
        $this->fixturesDir = __DIR__ . '/../fixtures/translations';
    }

    #[DataProvider('dataProvider')]
    public function testMessageFiles(string $file): void
    {
        $this->assertFileExists($file);

        // Test file format.
        $messages = require $file;
        $this->assertIsArray($messages);

        // Test message format.
        foreach ($messages as $key => $value) {
            $this->assertIsString($key);
            $this->assertIsString($value);

            // Test ICU format.
            $formatter = new MessageFormatter('en', $value);
            $this->assertNotFalse(
                $formatter,
                "Invalid ICU format in message: $value"
            );
        }
    }

    public function provideMessageFiles(): array
    {
        return [
            'en messages' => [$this->fixturesDir . '/messages/en.php'],
            'es messages' => [$this->fixturesDir . '/messages/es.php'],
            'en errors' => [$this->fixturesDir . '/errors/en.php'],
            'es errors' => [$this->fixturesDir . '/errors/es.php'],
        ];
    }
}
```

Testing Translation Consistency

```
class TranslationConsistencyTest extends TestCase
{
    private array $locales = ['en', 'es'];

    private array $domains = ['messages', 'errors'];

    private MessageProviderInterface $provider;

    protected function setUp(): void
    {
        $this->provider = new PhpMessageProvider(
            __DIR__ . '/../fixtures/translations'
        );
    }

    public function testAllLocalesHaveSameKeys(): void
    {
        foreach ($this->domains as $domain) {
            $baseMessages = $this->provider->getMessages('en', $domain);
            $baseKeys = array_keys($baseMessages);

            foreach ($this->locales as $locale) {
                if ($locale === 'en') {
                    continue;
                }

                $messages = $this->provider->getMessages($locale, $domain);
                $keys = array_keys($messages);

                $this->assertEquals(
                    sort($baseKeys),
                    sort($keys),
                    "Missing translations in $locale for domain $domain"
                );
            }
        }
    }

    public function testIcuParametersConsistency(): void
    {
        foreach ($this->domains as $domain) {
            $baseMessages = $this->provider->getMessages('en', $domain);

            foreach ($this->locales as $locale) {
                if ($locale === 'en') {
                    continue;
                }
```

```
                $messages = $this->provider->getMessages($locale, $domain);

                foreach ($baseMessages as $key => $baseMessage) {
                    $message = $messages[$key];

                    // Extract parameters from ICU messages.
                    preg_match_all('/{(\w+)}/', $baseMessage, $baseParams);
                    preg_match_all('/{(\w+)}/', $message, $params);

                    $this->assertEquals(
                        sort($baseParams[1]),
                        sort($params[1]),
                        "Parameters mismatch in key '$key' for locale $locale"
                    );
                }
            }
        }
    }
}
```

Testing Custom Providers

```
class CustomProviderTest extends TestCase
{
    public function testProvider(): void
    {
        $provider = new CustomProvider();

        // Test basic functionality.
        $messages = $provider->getMessages('en');
        $this->assertIsArray($messages);

        // Test locales.
        $locales = $provider->getAvailableLocales();
        $this->assertNotEmpty($locales);

        // Test domains.
        $errors = $provider->getMessages('en', 'errors');
        $this->assertIsArray($errors);
    }

    public function testErrorHandling(): void
    {
        $provider = new CustomProvider();

        $this->expectException(RuntimeException::class);
        $provider->getMessages('invalid-locale');
    }
}
```

Test Helpers

```
trait TranslationTestTrait
{
    private function createTestTranslator(): TranslatorInterface
    {
        return new class implements TranslatorInterface {
            public function trans(
                ?string $id,
                array $parameters = [],
                string $domain = null,
                string $locale = null
            ): string {
                return strtr($id, $parameters);
            }

            public function getLocale(): string
            {
                return 'en';
            }
        };
    }

    private function assertTranslationEquals(
        string $expected,
        TranslatableException $exception,
        string $message = ''
    ): void {
        $translator = $this->createTestTranslator();
        $this->assertEquals(
            $expected,
            $exception->trans($translator),
            $message
        );
    }
}
```

Common Patterns

1. **Test Exception Factory Methods**

```
public function testExceptionFactory(): void
{
    $exception = ValidationExceptionFactory::required('email');
```

```
    $this->assertInstanceOf(ValidationException::class, $exception);
    $this->assertEquals(
        'The field email is required.',
        $exception->getMessage()
    );
}
```

2. **Test Translation Fallbacks**

```
public function testFallbackTranslation(): void
{
    $translator = new Translator(
        $this->provider,
        'fr',
        ['es', 'en']
    );

    $exception = new ValidationException('test.message');

    // Should fallback to English.
    $this->assertEquals(
        'Test message',
        $exception->trans($translator)
    );
}
```

3. **Test ICU Edge Cases**

```
public function testComplexIcuPattern(): void
{
    $exception = new ValidationException(
        '{count, plural, =0{Empty} one{# item} other{# items}}',
        ['count' => 0]
    );

    $this->assertEquals('Empty', $exception->getMessage());
}
```

Remember:

- Always test both translated and untranslated scenarios.
- Test parameter substitution.
- Verify ICU message formatting.
- Check translation consistency across locales.

- Test error handling.
- Use data providers for multiple test cases.

Last updated on 21/08/2026