

Lint

Lint

Anonymous · 08/10/2026

Lint

Derafu\Translation\Lint has tools that read the code of a package to check that everything it shows to people can be translated. They are for tools and tests, not for the code that runs (see [Code Quality](#)). They need `nikic/php-parser`, which is optional: install it as a development dependency.

They read the code, they never run it. That is what makes them useful and what limits them: they can only check what can be known by reading. This page says how to write the code so that it can be checked, what breaks the check, and what the tools can not see.

If the conventions of this page are not followed, the package can not be validated. The tools do not guess: a message or an exception that can not be read is reported, and the test of the package fails until the code is written in a way that can be read (or the case is fixed explicitly in the test, see [Messages that can not be read by nature](#)). Nothing is skipped silently.

The audit

TranslationAudit does everything in one call: it finds the messages that the code builds, checks them against the catalogues of the package, and finds the exceptions that are not translatable. The test of a package is a few lines:

```
use Derafu\Translation\Lint\TranslationAudit;

final class TranslationMessagesTest extends TestCase
{
    public function testThePackageIsTranslated(): void
    {
        $report = (new TranslationAudit())->audit(
            dirname(__DIR__, 2) . '/src',
            new MyPackageTranslationResourceProvider(),
        );

        $this->assertFalse($report->nothingFound);
        $this->assertSame([], $report->describe($report->dynamicMessages));
        $this->assertSame([], $report->describe($report->missingTranslations));
    }
}
```

```

        $this->assertSame([], $report->describe($report->notUsedBySources));
        $this->assertSame([], $report->describe($report->notTranslatable));
    }
}

```

`audit()` takes:

Argument	What it is
<code>\$directory</code>	The code to read (every <code>.php</code> file, recursively).
<code>\$providers</code>	The catalogues of that code : one <code>TranslationResourceProviderInterface</code> or an iterable of them.
<code>\$locale</code>	The locale whose catalogue is checked. <code>es</code> by default.
<code>\$allowedThrowables</code>	Classes of exceptions that are not translatable and are allowed, by their full name.
<code>\$messageMethods</code>	The methods of the package that receive the id of a message (<code>MessageMethod</code>).

Use the catalogues of **that** code. A message of a package is translated by the catalogue of that package, and which one belongs to which is something only you know. That is why the tools do not decide.

The report

`TranslationAuditReport` only has facts. It never says that something is a problem: that is for your test to say, by asserting that the lists it cares about are empty.

Property	The fact
<code>dynamicMessages</code>	Messages that are not a literal, so they can not be checked.
<code>missingTranslations</code>	Messages that have no entry in the catalogues, in their domain.
<code>notUsedBySources</code>	Entries of the catalogues that no message of the code that was read uses.
<code>notTranslatable</code>	Exceptions that are thrown or declared and are not translatable, and were not allowed.
<code>nothingFound</code>	<code>true</code> when no message was found at all (the directory is wrong, or the code has none).

`describe()` turns a list of findings into lines, one each, for the message of a failed test: relative path and line, and what it is ("Message {name}." [errors], Class extends Parent, or the function and call for a message that can not be read). When a test fails, the failure says exactly where.

`notUsedBySources` is relative to what was read: an entry can be used by something that was not, for example a template. Decide for each package whether to assert it empty.

Conventions

These are the rules that make a package checkable.

Exceptions

1. **Every exception that the package throws, or declares, is translatable.** Extend one of the `Translatable*` classes of this package, or use `TranslatableExceptionTrait` and implement `TranslatableInterface` ([Exceptions](#)). That includes the `\Error` family: a `TypeError` or a `ValueError` has its translatable class.

2. **Write the message as the English text, in the form of an array**, with named parameters:

```
throw new NotFoundException(['Resource "{name}" not found.', 'name' => $name]);
```

The text is the id of the translation and the fallback. The Spanish (or any other language) is only in the catalogue, never in the code.

3. **An external text goes inside a phrase of your own**, in English, as a parameter:

```
throw new MyException(['A problem happened: {message}', 'message' =>
    $e->getMessage()]);
```

The text of another library or of PHP (`json_last_error_msg()`, an exception of a vendor) can not be translated by you, but the sentence around it can. If there is no better sentence, use the generic one with the key `message`: `['{message}', 'message' => $text]`.

4. **To add context to a message that is its own, nest it:** a parameter can be another `TranslatableMessage` (or a translatable exception), and both are translated, the sentence and the one inside.
5. **An exception that must be thrown as it is** (because a contract requires that class, for example a Symfony interface that asks for `CommandNotFoundException`) is allowed explicitly with `$allowedThrowables`. It is a decision of the package, written in its test, and not a default.

Messages

6. **Write the id as a literal string.** The tool reads the text, so the text has to be there.
7. **Write the domain as a literal**, or do not write it (the exception has its `$defaultDomain`).
8. **Use ICU named parameters in the message** and give the values in the array. Do not build

the text with `sprintf()`, concatenation or an interpolated string.

9. **A catalogue entry has the same id as the message**, in the domain of the exception. The catalogue is in `resources/translations/<domain>+intl-icu.<locale>.php` and has a `TranslationResourceProvider` that the package registers (tagged `derafu_translation.resource_provider`).

Helper methods

A class that has its own method to translate (for example `private function trans(string $message, array $parameters = [])`) is read in a different way: inside the method the id is a variable, so the tool reads the **calls** to the method, where the literals are. For that, declare it with `MessageMethod` ([below](#)).

10. **Do not name a helper like the method of another object** that you also call in the same code. The tool finds calls by the name of the method. A helper called `trans` and a call `$translator->trans(...)` on another object look the same when the receiver can not be known by reading, and the second is reported as a message that can not be read. Give the helper a name that is yours (`translate`, `message`, `text`) or call the standard form.
11. **If the id of a helper comes from a constant, a map or a variable**, the tool can not read it. Write each call with a literal (`$this->trans('Copy value')`), or use `match` with a literal in each branch.

What not to do

Each of these makes the code impossible to check by reading, and the package can not be validated while it is there:

Do not	Why	Do
<code>throw new MyException(sprintf('Failed %s.', \$x))</code>	The id changes with every value: it can not have an entry.	<code>['Failed {x}.', 'x' => \$x]</code>
<code>throw new MyException("Failed { \$x }.")</code>	Same.	<code>['Failed {x}.', 'x' => \$x]</code>
<code>throw new MyException('Failed: ' . \$e->getMessage())</code>	Concatenation, same.	<code>['Failed: {message}', 'message' => \$e->getMessage()]</code>
<code>throw new MyException(\$message) where \$message comes from outside</code>	The tool can not know it.	Pass the literal where the exception is made.
<code>new TranslatableMessage(\$text, [], \$domain) with \$domain a variable</code>	The domain can not be read, so the entry can not be found.	Write the domain as a literal.

Do not	Why	Do
<code>throw new \RuntimeException(...), \InvalidArgumentException, \LogicException, \JsonException, \TypeError, \ValueError, \BadMethodCallException, ...</code>	Not translatable.	The <code>Translatable*</code> class (same name, same parent).
An exception of your own that extends <code>\Exception</code> (or any native one)	Its declaration is reported, and so is every <code>new</code> of it.	<code>extends TranslatableException</code> (or the one that fits).
A helper whose id is a variable and is not declared	Its calls are not messages for the tool, so the entries look unused.	Declare it with <code>MessageMethod</code> .
An exception thrown only to be caught in the same method	The tool sees the <code>throw new</code> and can not see the <code>catch</code> .	Do not throw it (return, or use a condition), or allow the class.
Spanish (or another language) in the message in code	The id is English: it is what shows when there is no translation.	English in code, Spanish in the catalogue.

The same exception, native and translatable

The translatable exceptions are named like the native ones with the prefix `Translatable`. In a file that used `InvalidArgumentException`, the cheapest change is an alias, with no other line touched:

```
use Derafu\Translation\Exception\Logic\TranslatableInvalidArgumentException as
InvalidArgumentException;
```

The tool tells them apart by the class that the name resolves to, not by the text, so the alias is read as the translatable one.

Messages that can not be read by nature

Some code can not have a literal. The clearest case is `TranslatableExceptionTrait`, which wraps in a `TranslatableMessage` the message that whoever throws the exception gives it. Those are reported as dynamic, correctly.

You fix them in the test, by what they are, not by where they are. The identity of a dynamic message is the function and the whole call:

```
$this->assertSame(
    [
        'Derafu\\Translation\\Trait\\TranslatableExceptionTrait::normalizeMessage: '
```

```

        . 'new \\Derafu\\Translation\\TranslatableMessage($message, [],
$this->defaultDomain, $this->defaultLocale)',
    ],
    array_map(fn (MessageReference $reference) => $reference->identity(),
$this->dynamicMessages)
);

```

It does not change when other lines of the file do. It does change when that call changes, and it fails if it disappears or if a new one appears anywhere else, so a new message that can not be read never goes unnoticed. If one is fixed in the code, take it out of the list.

Do not fix a dynamic message by pinning the file or the line: a new one in the same file would be hidden.

Helper methods: `MessageMethod`

```

use Derafu\Translation\Lint\MessageMethod;

$report = (new TranslationAudit())->audit(
    $src,
    new MyPackageTranslationResourceProvider(),
    messageMethods: [
        new MessageMethod(ContactController::class, 'trans', domain: 'contact-form'),
    ],
);

```

Argument	What it is
<code>\$class</code>	The class that has the method (use <code>::class</code>). Calls on its subclasses count. Whether the class and the method exist is checked, and a mistake raises an <code>InvalidArgumentException</code> and does not hide the check.
<code>\$method</code>	The name of the method.
<code>\$domain</code>	The domain of its messages. If it is not given, it is the default of <code>TranslatableMessage</code> (messages).
<code>\$id</code>	Where the id is: the position (starting at 0) or the name of the argument. 0 by default.
<code>\$domainArgument</code>	Where the domain is, if the call can give its own.

The tool then finds each `$this->trans('...')` (and `self::`, `static::`, `parent::` and `Class::` calls) as a message with its id and domain, and checks it like the others. The method's own body, where the id is a variable, is not reported.

The pieces

`TranslationAudit` is made of two scanners that can be used on their own.

MessageReferenceScanner

Finds every message that the code builds to be translated: the one given to a translatable exception and to new `TranslatableMessage(...)`, and the calls of the methods declared with `MessageMethod`.

```
use Derafu\Translation\Lint\MessageReferenceScanner;

$scanner = new MessageReferenceScanner(/* list<MessageMethod> */);

foreach ($scanner->scanDirectory($src) as $reference) {
    $reference->id;           // 'Resource "{name}" not found.', or null when it is not
    a literal.
    $reference->domain;       // 'errors', or null when it is not a literal.
    $reference->class;        // The exception (or TranslatableMessage) that is built.
    $reference->file;
    $reference->line;
    $reference->function;     // 'Class::method', or 'Class::method::{closure}'.
    $reference->expression;   // The whole call, when the message can not be read.
    $reference->isDynamic();
    $reference->identity();   // function + whole call, or the id.
}
```

`scanFile()` does the same for one file. `scanDirectory()` takes every `.php` file of a directory and its subdirectories, in order of file and then of line.

What it finds:

- **The message of a translatable exception:** a string or the first element of an array, also as a named argument (`message:`).
- **The domain:** the default of its `$defaultDomain` property (errors unless the class sets another one). For new `TranslatableMessage(...)` it is the third argument, or `messages`.
- **Both branches of a ternary** (`$a ? 'One.' : 'Two.'`) as two messages. In `$a ? : 'Two.'` the left side is dynamic and the right is a literal.
- **What a class builds for itself:** `parent::__construct(...)`, `new static(...)` and `new self(...)` in its factories, and the default of its `$message` parameter.
- **Anonymous classes** that extend a translatable one.

A message that is a parameter of a constructor and is only passed on to `parent::__construct($message)` is not a reference: it is the message of whoever calls the constructor, and that call is found where it is made. An empty id is ignored.

ThrowReferenceScanner

Finds the exceptions that are not translatable: every `new` of a throwable that does not implement `TranslatableInterface` (whether it is thrown there, later, or given back by a factory), and every class that is declared extending one.

```
$references = (new ThrowReferenceScanner())->scanDirectory($src);

$references[0]->class;           // 'RuntimeException'
$references[0]->extends;        // null for a throw, or the parent for a declaration.
$references[0]->isDeclaration();
$references[0]->loadable;       // false if the class could not be loaded.
```

A `throw new` of a class that can not be loaded, or of a class that is a variable, is reported as unknown: what is thrown is an exception, and finding nothing would look like a clean result.

How they work

- They read the code, so comments and strings that only look like a `new` are ignored, and the references are found wherever a `new` can be (closures, arrow functions, arguments, anonymous classes).
- They tell the classes apart by what they are (a throwable that implements `TranslatableInterface`), not by their name. Classes must be loadable (autoloaded), because they are checked by reflection.
- A file that can not be parsed raises the parser's `PhpParser\Error`. A directory or file that does not exist raises an `InvalidArgumentException`. They fail loudly: a clean result must never be the result of not having read.

Known limits

These are things the tools can not see. They are not bugs: they are what reading code, without running it, can know. Knowing them is part of using the tools.

1. **throw \$e of an exception that was not made in the code that is read.** An exception that is caught and thrown again, or one that a function returns, is not seen: without the types of the code it is not known which class it is. Only a `new` is seen.
2. **A thrown and caught exception looks like a thrown one.** The tool sees the `throw new` and not the `catch`, so an exception that is thrown to be caught in the same method (for control flow) is reported. Rewrite the code without it, or allow the class with `$allowedThrowables`.
3. **Helper methods are found by their name and their receiver.** The receiver is known when it is `$this`, `self`, `static`, `parent` or the name of a class. On any other receiver (a variable, a property, the result of a call) it is not known by reading, and if the method has the name of a declared helper it is reported as a message that can not be read. That is what makes a public helper noisy when other objects have a method with the same name. Rename the helper, or call

it in a way that the receiver is known.

4. **Calls of a helper on another instance are not seen**, for the same reason:
`$other->trans('Text')` is not read as a message of the helper of the class of `$other`.
5. **A helper whose id is not a literal in the call** (a constant, a map, a variable) is reported as a message that can not be read. The id has to be where the call is.
6. **notUsedBySources is relative to what was read**. Messages that are used only by something that is not PHP code in the audited directory, for example a template, look unused. If the package has such messages, do not assert this list empty.
7. **It checks that the entry exists, not that it is good**. A catalogue entry with the wrong translation, a missing parameter or a different meaning passes the audit. It is for a person to review.
8. **Catalogues are shared by domain**. Two packages that use the domain `errors` and the same id with different translations override each other, and no test of a single package can detect it. The ids are English phrases, so it is unlikely, but it can happen.
9. **TranslatableMessage::trans() only knows the entries of translators that expose their catalogues** (TranslatorBagInterface, like Symfony's). With another translator, a message without an entry is whatever that translator returns.
10. **Native classes with their own constructor are not covered**. `ErrorException` takes `severity`, `filename` and `line`, which `TranslatableExceptionTrait` can not use, so there is no translatable version of it.
11. **Anything that is not PHP code is out of reach**: texts in templates, in configuration, in JavaScript or in the database.
12. **Reading, not types**. The tools do not infer types. A class that is not loadable (a missing dependency) is not known to be translatable, and it is reported.

Last updated on 08/10/2026