

Error Handling

Error Handling

Anonymous · 08/10/2026

Error Handling

Any exception thrown while a request is processed, whether by a controller, a middleware or the router itself, is caught and turned into a response that follows [RFC 7807](#) (Problem Details). You do not write a `try/catch` for that: throw, and the response is built from the exception.

```
public function show(string $id): array
{
    $item = $this->repository->find($id)
        ?? throw new NotFoundException(['Item {id} not found.', 'id' => $id]);

    return $item->toArray();
}
```

How an Exception Becomes a Response

1. The `RequestHandler` catches the exception.
2. `ProblemFactory` builds a `ProblemDetail` from it and from the request.
3. `ProblemHandler` renders the problem in the format the client prefers.

The HTTP status

The status is the first of these that applies:

1. `getStatus()`, if the exception implements `HttpExceptionInterface`.
2. The exception code, if it is a valid HTTP status code.
3. 404 for a `RouteNotFoundException`.
4. 500 for anything else.

The problem

Field	Value
<code>type</code>	<code>getUriReference()</code> of an <code>HttpExceptionInterface</code> , otherwise <code>about:blank</code>
<code>title</code>	<code>getTitle()</code> of an <code>HttpExceptionInterface</code> ; for any other exception, the reason phrase of the status. It is translated when there is a translator

Field	Value
status	The HTTP status code
detail	The message of the exception. If the exception is translatable, it is translated when there is a translator
instance	The path of the request
extensions	timestamp, environment, debug, context, headers and throwable

The title is the one the problem is created with, also when the type is `about:blank`: with `about:blank` it should be the reason phrase of the status, which can be localized ([RFC 9457](#), section 4.2.1). Without a title, the reason phrase of the status is used.

`throwable` carries the exception, with its trace, only when `kernel.debug` is on. Otherwise it is `null`, so production responses do not expose internals.

Translation

`ProblemFactory` translates the `title` and the `detail` of the problem when it has a translator (`Symfony\Contracts\Translation\TranslatorInterface`). It receives it as an optional argument of its constructor: when the container injects one the texts are translated, and without one they are left as they are written.

- The **detail** is the message of the exception. It is translated when the exception is translatable (it implements `TranslatableInterface`, see [HTTP Exceptions](#)), with the catalogue of the domain the exception uses. Any other exception keeps its message.
- The **title** is its own translation id, in the `errors` domain, the same one the messages of the exceptions use. The reason phrases of the statuses (`Not Found`, `Forbidden`...) have their Spanish translation in the catalogue of the package. The title of your own `HttpExceptionInterface` needs an entry in your catalogue; a text without an entry is kept as it is.

The catalogue of the package is provided by

`Derafu\Http\Translation\HttpTranslationResourceProvider`. It is registered, with the other services of the package, by `resources/config/http-services.yaml`, which you import from your `services.yaml`:

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

To register only the catalogue, tag the provider with `derafu_translation.resource_provider`, as [derafu/translation](#) explains.

Response Formats

The format follows the content negotiation of the request. See [Requests and Responses](#).

JSON

An API request, or a client that asks for JSON, gets the problem as JSON with the status of the error:

```
{
  "type": "about:blank",
  "title": "Not Found",
  "status": 404,
  "detail": "No route found for \"/api/mustFail\".",
  "instance": "/api/mustFail",
  "extensions": {
    "timestamp": "2025-02-20T22:04:25+00:00",
    "environment": "prod",
    "debug": false,
    "context": [],
    "headers": [],
    "throwable": null
  }
}
```

HTML

For a browser, the library renders an error page. The pages are given to `ProblemHandler` as a map from the status (404, 500...) to a handler, and `default` for the statuses that have none. A handler is the same kind of string as the handler of a route: the path of a template or a `Controller::action`. The map is the parameter `http.error_pages`, which you set in your `services.yaml` (with [derafu/foundation](#) it already has `templates/error.html.twig` as default):

```
parameters:
  http.error_pages:
    # Specific page for a 404.
    404: '%kernel.project_dir%/templates/error404.html.twig'
    # The page for any other status. It can be a controller.
    default: 'App\Controller\ErrorController::show'
```

The page of the status is used first, and then `default`; if a page fails the next one is tried. When the handler returns plain content, like a rendered template, the page is sent with the status of the problem. A handler that returns its own `Response` is sent as is.

The error pages are **not routes**: a route is a resource with a URL that anyone can ask for, and an error page only makes sense as the answer to an error. `/error` is a path like any other that does not exist.

The problem is passed to the handler inside the `context` array, under the `error` key. In a Twig template it is available as `context.error`:

```
<h1>{{ context.error.status }}</h1>
<h2>{{ context.error.title }}</h2>
<p>{{ context.error.detail }}</p>

{% if context.error.debug %}
    <pre>{{ context.error.throwable.traceAsString }}</pre>
{% endif %}
```

The keys of the map must be a status from 400 to 599 or `default`, and a handler a text that is not empty (or a `Closure`, for a service that is built in PHP): anything else is an error of the configuration, thrown when the handler is created.

If there are no pages, or all of them fail, the problem is sent as a Markdown text instead. That is the last resort for a serious failure, not the answer to a page that is missing. With `kernel.debug` on, the text says which pages failed and why (`## Error pages that failed`).

Markdown

A request for a URL that ends in `.md` gets the problem as Markdown text, and so does the HTML fallback above. With `kernel.debug` on it includes the exception and its trace.

HTTP Exceptions

To control the status, title, type, context and headers of an error, implement `Derafu\Http\Contract\HttpExceptionInterface`:

Method	Returns
<code>getStatus()</code>	The <code>HttpStatus</code> of the response
<code>getTitle()</code>	A short summary of the type of problem
<code>getUriReference()</code>	A URI that identifies the type of problem
<code>getContext()</code>	Extra data, sent in <code>extensions.context</code>
<code>getHeaders()</code>	Headers added to the response

The interface extends `TranslatableInterface` from `derafu/translation`, so the message is written in its `[message, ...params]` form, where the parameters are interpolated into `{placeholders}`.

```
use Derafu\Http\Contract\HttpExceptionInterface;
use Derafu\Http\Enum\HttpStatus;
use Derafu\Translation\Exception\Core\TranslatableRuntimeException;
```

```

class NotFoundException extends TranslatableRuntimeException implements
HttpExceptionInterface
{
    public function getUriReference(): string
    {
        return 'https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404';
    }

    public function getTitle(): string
    {
        return 'Not Found';
    }

    public function getStatus(): HttpStatus
    {
        return HttpStatus::NOT_FOUND;
    }

    public function getContext(): array
    {
        return [];
    }

    public function getHeaders(): array
    {
        return [];
    }
}

```

Provided exceptions

- `BadRequestException`: a 400, for a request that can not be attended because of what the client sent (a field that is missing, a file that is not there). It is an `InvalidArgumentException`, so a service that validates what it receives can throw it without knowing about HTTP, and the callers that already catch the invalid argument keep working. Its message is `Bad request`. unless you give another one.
- `TooManyRequestsException`: a 429. It accepts the headers to send as its last constructor argument. `ThrottleMiddleware` uses it to send `Retry-After` and the `X-RateLimit-*` headers. See [Middleware](#).
- `InvalidPathException` (of `derafu/routing`): a 400, thrown by the router when the path of the request has no safe canonical form (a `..` segment, an escaped slash, a control character). See [Middleware](#).
- `ResponseSerializationException`: thrown when a response can not be encoded as JSON and is not a string.
- `DispatcherException`: thrown when the handler of a route is not valid.

Notes

- **A problem is built only when something fails.** A request that succeeds does not go through this code.
- **Headers always apply.** The headers of an `HttpExceptionInterface` are added to the response whatever the format.
- **Keep `kernel.debug` off in production.** It is what hides the exception and its trace from the response.

Last updated on 08/10/2026