

Results and Failures

Results and Failures

Anonymous · 10/10/2026

Results and Failures

Handling Failure: `ProblemDetail`

An RFC 7807-shaped, transport-agnostic problem description — `type/title/detail/instance` at the top level, everything else namespaced under `extensions`:

```
$problem->getDetail();           // The exception's own message.
$problem->getInstance();         // The failed request's id:
"billing.invoice.builder::build".
$problem->getThrowable()->getClass(); // e.g. "RuntimeException" – only exposed when
debug is true.
$problem->toArray();
```

```
[
  'type' => 'about:blank',
  'title' => 'RuntimeException',
  'detail' => 'Something went wrong while running the operation.',
  'instance' => 'billing.invoice.builder::build',
  'extensions' => [
    'timestamp' => 1755500000.123456, // Unix epoch, seconds – same instant and
representation as `ExecutionMetadata::getTimestamp()`.
    'data_type' => null,              // Always null on a failure – there is no
value to describe.
    'environment' => 'prod',
    'debug' => false,
    'context' => [],
    'throwable' => null, // hidden outside debug mode.
  ],
]
```

The wrapped `SafeThrowable` actively scrubs sensitive data before it ever gets this far: trace frame `args` are stripped, and absolute file paths are rewritten relative to a project directory (`"project_dir:src/Foo.php"` instead of `/home/user/project/src/Foo.php`) — neither call arguments nor local filesystem layout leak to whatever is on the other side of the boundary.

Execution Metadata: `ExecutionMetadata`

Every `OperationResultInterface` — success or failure alike — also carries `getMetadata()`: `ExecutionMetadataInterface`, statistics about the dispatch that produced it. A consumer decides whether, and how, to use these; this package only collects them:

```
$metadata = $result->getMetadata();

$metadata->getStartedAt();          // "2026-01-20T10:00:00+00:00"
$metadata->getFinishedAt();
$metadata->getTimestamp();          // The same instant as getFinishedAt(), as a Unix
epoch float — flexible to reuse (sorting, arithmetic), unlike the DATE_ATOM string.
$metadata->getRealTime();           // Seconds, wall-clock — the "real" of `time`.
$metadata->getUserTime();           // Seconds of CPU in user mode — the "user" of
`time`.
$metadata->getSystemTime();         // Seconds of CPU in kernel mode — the "sys" of
`time`.
$metadata->getMemoryUsed();         // Bytes, a delta — can be negative if the GC freed
more than this dispatch allocated.
$metadata->getPeakMemory();         // Bytes, the whole process's peak up to this point —
stable against that same GC noise.
$metadata->getPid();
$metadata->getLoadAverage1Min();    // Plus 5/15-minute variants — tells "slow because of
this operation" apart from "slow because the system itself was saturated."
```

Assumes Linux/macOS: built on `getrusage()` and `sys_getloadavg()`, neither of which exists on Windows — no Windows support is offered.

`TypedDispatcher` and `SafeDispatcher` each measure their own scope independently, never reusing the other's numbers: `TypedDispatcher`'s metadata covers only resolving parameters and invoking the worker (via `DirectDispatcher`); `SafeDispatcher`'s covers that plus serializing the result on success, or everything up to the moment it caught the exception on failure.

`ProblemDetailInterface::getTimestamp()` (above) does reuse `ExecutionMetadata`'s own reading rather than taking a second, independent one, so both always agree on the exact same instant for the same dispatch.

What Kind of Value Was Returned: `getDataType()`

Alongside `getValue()`, a successful `OperationResultInterface` also carries `getDataType(): ?string` — the type of the value *before* `SafeDispatcher` serializes it (`get_class()` for an object, `gettype()` for a scalar/array), e.g. "App\Entity\Invoice" or "integer". null on a failure, since there is nothing to describe.

```
$result->getDataType(); // e.g. "App\Entity\Invoice" — even though getValue() is
already a plain array.
```

This exists because the information is only available at the exact moment of dispatch: once `SafeDispatcher` has serialized a domain object into a plain array, its original class is gone for good — a transport built on top (like [Backbone API](#) or [Backbone Console](#)) cannot recover it afterward, so `TypedDispatcher` captures it up front and `SafeDispatcher` carries it through unchanged.

Turning Array Data Into Real Objects

A parameter typed as a class or interface doesn't have to arrive pre-built — a plain array (or string, for things like base64-encoded certificates) is deserialized on the way in:

- Any class exposing a static `fromArray(array $data): self` works with **zero registration**, via `FromArrayDeserializer` (the conventional fallback).
- A specific class can instead get its own `DeserializerInterface` registered on `ObjectFactoryRegistry`, which takes priority over the fallback — useful when construction isn't a plain `fromArray()` (loading a certificate from either raw data or a key pair, for example).
- Union-typed parameters (`A|B`) try each candidate class in order.

```
$resolver = new Resolver(  
    new Inspector(),  
    new Caster(new ObjectFactoryRegistry(  
        deserializers: [Caf::class => new CafDeserializer()], // explicit, takes  
priority.  
        fallback: new FromArrayDeserializer(),                // used for everything  
else.  
    )),  
    new Validator(),  
);
```

On the way out, `Serializer` mirrors this: arrays recurse, `JsonSerializable` objects recurse into `jsonSerialize()`, objects with a `toArray()` recurse into that — so a nested domain object graph comes back from `SafeDispatcher` as plain, nested arrays.

Last updated on 10/10/2026