Testing a Dispatcher

Testing a Dispatcher

Anonymous · 15/11/2026

Testing a Dispatcher

`Derafu\BackboneDispatcher\Testing` is a ready harness for the tests of any dispatcher built on this package, with nothing tied to a domain. It has two abstract PHPUnit test cases that already contain the tests. A dispatcher extends them, implements a few `protected static` hooks that return its real services, and gets:

- `DispatcherConformanceTestCase`: rules that every dispatcher follows, checked from what the explorer discovers. No data and no files.
- `ScenarioTestCase`: the business cases of the dispatcher, written as YAML files that dispatch operations or build objects with the real services and say what they expect.

Both go through the real `SafeDispatcher`, `SafeExplorer` and `ObjectFactory` of the application: there are no mocks in the chain. How the application exposes its services to the tests (for example a public alias of the object factory in a test container) is up to each application.

PHPUnit, `symfony/yaml` and `mtdowling/jmespath.php` are not dependencies of the package. They are `suggest`ed: install them as `require-dev` where the harness is used. A feature whose package is missing throws `MissingOptionalDependencyException` that names the package.

Conformance suite

```
use Derafu\BackboneDispatcher\Testing\DispatcherConformanceTestCase;

#[CoversNothing]
class ConformanceTest extends DispatcherConformanceTestCase
{
    protected static function explorer(): SafeExplorerInterface
    {
        return Bootstrap::bootExplorer('test', true);
    }

    protected static function objectFactory(): ObjectFactoryInterface
    {
        return TestServices::objectFactory();
    }
}
```

Every rule is one test method that walks `explorer()->tree()`, collects every violation and fails with all of them, one per line (`id: reason`):

Test	Rule
`testNamesFollowTheConvention`	The name of every package, component, worker and operation (the last segment of its id) matches `namePattern()`. The parameter names match `parameterNamePattern()` when it is not `null`.
`testOperationsAreDocumented`	Every operation has a non-empty `summary` and `description`.
`testParametersHaveExamples`	Every parameter has an `example`.
`testExamplesAreEncodableAsJson`	Every example is made of `null`, scalars, arrays and backed enum cases, which is what an HTTP client sends. Any other object is a violation.
`testObjectParametersHaveADeserializer`	Every class in a parameter type (each candidate of a union, without `?` or `[]`) is a backed enum or satisfies `ObjectFactoryInterface::supports()`.
`testReturnTypesAreSerializable`	Every candidate of a return type is a built-in type that serializes as it is, an enum, or a class that implements `JsonSerializable` or has a `toArray()`. An intersection type is serializable when any of its parts is. `mixed` and `object` are violations because nothing says what comes back.

The style of the names is a decision of each project, so it is a hook:

```
protected function namePattern(): string;           // snake case by default.
protected function parameterNamePattern(): ?string; // null by default: not checked.
```

A violation that cannot be fixed yet is listed in `exclusions()` with the reason. The key is the rule (the name of its test without `test`, in snake case) and the value maps the id that has the violation to the reason why it is allowed:

```
protected function exclusions(): array
{
    return [
        'parameters_have_examples' => [
            'billing.document.parser::parse' => 'The example depends on the format.',
        ],
    ];
}
```

An exclusion whose id no longer has a violation fails the rule, so the list cannot go stale.

Scenarios

```
use Derafu\BackboneDispatcher\Testing\ScenarioTestCase;

#[CoversNamespace('App\Dispatcher')]
class ScenarioTest extends ScenarioTestCase
{
    protected static function scenariosDirectory(): string
    {
        return dirname(__DIR__) . '/fixtures/scenarios';
    }

    protected static function dispatcher(): SafeDispatcherInterface
    {
        return Bootstrap::boot('test', true);
    }

    protected static function objectFactory(): ObjectFactoryInterface
    {
        return TestServices::objectFactory();
    }

    protected static function fixtures(FixtureRegistry $registry): void
    {
        $registry->fixture('cal', static fn (): string => TestData::calInBase64());
    }
}
```

Every `*.yaml` file under `scenariosDirectory()` (recursively, in order of path) is one test, so adding a case is adding a file. A directory without scenarios is an error, not a test that passes by running nothing. `serializer()` returns a `Serializer` by default and can be overridden.

PHPUnit needs the coverage a scenario exercises to be declared (`#[CoversNamespace]`, or `#[CoversNothing]`), because a scenario goes through whatever the operations it dispatches go through.

Format

```
description: A CAL is built again from the XML of its own serialization.
requires: [docker]            # Optional. A missing requirement skips the scenario.
steps:                        # Ordered. The name is [a-z][a-z0-9_]* and is what !ref
uses.
  load:
    deserialize:
      type: Vendor\Contract\CalInterface   # Built with
ObjectFactoryInterface::create().
      data: !fixture cal                   # A string, a map or a tag.
    expect:
```

```
      id: CAL-1
      call:
        getName(): A name
  again:
    deserialize:
      type: Vendor\Contract\CalInterface
      data: !ref load.value.xml
    expect:
      '@': !ref load.value
  send:
    dispatch:
      operation: package.component.worker::operation   # Dispatched with
SafeDispatcherInterface.
      parameters:
        name: Ana
    failure:
      exception: Vendor\Exception\Foo
      message: text that the message contains
```

A step has exactly one of `dispatch` or `deserialize`, and at most one of `expect`, `failure` or `either`. Without any of them it only has to succeed. A key that the format does not know, a missing key or a wrong type is an `InvalidScenarioException` that names the file and the key.

`dispatch` gives as its value what `SafeDispatcher` answers, already serialized. If the operation fails, the step has the problem: its type, title (the class of the exception) and detail.

`deserialize` gives as its value the serialization of the object that the object factory builds, and the object itself for `call`. The object has to be an instance of `type` (or the data an array when one candidate of a union is `array`). A rejection of the object factory is the failure of the step.

`expect` is a map of JMESPath expressions to the expected values, evaluated over the serialized value, with strict equality. An expression gives what JMESPath gives: an empty list is an empty list and a list with one element is a list, not its element. `'@'` is the whole value, whatever it is, and any other expression needs the value to be an array. A map has to have the same keys in the same order, and `!any` or `!ref` can be nested inside maps and lists. In a `deserialize` step, `expect` can also have `call`, a map of calls on the object that serialization does not show:

```
expect:
  call:
    getName(): LibreDTE            # What the method returns (an object is compared by
its serialization).
    getSigner().rut: 11111111-1    # A JMESPath expression over the serialization of
what it returns.
    getSigner(): !instanceof Vendor\Signer
```

`failure` needs at least one of:

- `exception`: a class name, or a list of them. For a `deserialize` step, the exception is an instance of one. For a `dispatch` step, the title of the problem is one of those classes or a subclass.
- `caused_by` (`deserialize` only): a class that is the exception or is in its chain of previous exceptions. It checks a rejection that the object factory wraps in `NoDeserializerFoundException`.
- `message`: a text that the message of the exception (or the detail of the problem) contains.

`either` is `{expect: ..., failure: ...}` and passes when the step succeeds and matches `expect`, or fails and matches `failure`. It is for operations that call external services, whose answer is not deterministic, and each scenario that uses it says so in a comment.

Tags

Tag	Where	Meaning
`!fixture name` or `!fixture {name: n, args: {...}}`	`parameters`, `data`, and the `args` of another fixture	The value of the fixture `n`, built by its provider with the arguments.
`!ref expression`	The same places and the expected values	A JMESPath expression over the results of the previous steps. It starts with the name of a step and reads its `value` or its `problem`: `!ref load.value.xml`, `!ref list.value.items[0].id`. The step has to come before. A reference to a value that does not exist, or is `null`, is an error.
`!any`	Expected values	Matches any value that is not `null`.
`!instanceof FQCN`	The expected values of `call`	Matches an object of that class.

A tag in a place where it is not allowed is an `InvalidScenarioException`.

Fixtures and requirements

`fixtures()` registers the data that `!fixture` asks for in a `FixtureRegistry`:

```
$registry
    ->requirement('docker', static fn (): ?string => Docker::isRunning() ? null :
'Docker is not running.')
    ->fixture('book', static fn (array $args): string =>
TestData::book($args['type']))
    ->fixture('mail_server', static fn (): array => MailServer::start()->config(),
['docker'])
    ->onShutdown(static fn () => MailServer::stop());
```

A fixture is a name with a provider that receives the arguments of the tag, and whose result is not cached (a provider that is expensive to build does its own caching). A requirement is a named check of the environment that returns `null` when it is available, or the reason why it is not. The result of a

check is cached for the run. A scenario or a fixture whose requirement is missing is skipped with that reason, so a skip is always explicit and says why. `onShutdown()` hooks run, in reverse order, after the last scenario: a resource with a lifecycle registers its hook when it starts.

Round trips

What a dispatcher answers can be sent again. `expect` is mandatory in such scenarios because the equality of the two answers alone is not enough: a deserializer that ignores a piece of data from the first pass loses it in both, and they are equal. Feed back the part of the serialization that the deserializer accepts:

```
steps:
  first:
    deserialize: {type: Vendor\Contract\DocumentInterface, data: !fixture document}
    expect:
      id: abc
  second:
    deserialize: {type: Vendor\Contract\DocumentInterface, data: !ref first.value.xml}
    expect:
      '@': !ref first.value
```

The same works through the dispatcher, with two `dispatch` steps that receive the same type and return it.

Exceptions

`InvalidScenarioException` is an error of the test: the scenario cannot be run, so it never reaches the assertions. `FixtureUnavailableException` is not a failure: the test case turns it into a skipped test that says why. Both are translatable like the rest of the exceptions of the package.

Last updated on 15/11/2026