

Discovery

Discovery

Anonymous · 10/10/2026

Discovery

Discovery: Explorer and Inspector

Explorer walks the package registry (packages → components → workers → operations) — no `_links/HATEOAS` shaping, that's left to transports like [Backbone API](#). Inspector is the *only* class in the package that imports anything from `Reflection*`: every other collaborator that needs to know something about a class or a method — Resolver, Explorer, the operation policies — asks `InspectorInterface` for it instead of reflecting directly.

`getPackage()`, `getComponent()` and `getWorker()` include the real name/summary/description of each. name is read straight off the instance via `ServiceInterface::getName()` (from `derafu/backbone`, backed by its `#[Package]/#[Component]/#[Worker]` attributes) — a routing slug, never empty by omission, since it is what builds the discovery id. `summary/description` both come from the class' own PHPDoc: `summary` is always the PHPDoc's, verbatim (there is no `#[Worker(summary: ...)]` to prefer instead), while `description` prefers the same attribute's `description` argument first, falling back to the PHPDoc (its own `description`, then its `summary`) only when that argument was left unset — the same “explicit value, else PHPDoc” precedence `Inspector::getPublicMethods()` already gives an operation's `#[Operation]` attribute over its method's PHPDoc (see below), just one level up: a `#[Worker(...)]`-style attribute's `description` is just as easy to leave unset as an `#[Operation]`'s, and the class right below it almost always already documents itself. `getOperation(string $package, string $component, string $worker, string $operation)` looks up exactly one operation by name — unlike `getOperations()` (a listing, which silently omits what a policy rejects), a direct lookup throws: `OperationNotFoundException` if it does not exist, `OperationNotAllowedException` if the active policy rejects it.

Worth knowing: ids are dot-separated for the hierarchy (`"package"`, `"package.component"`, `"package.component.worker"`) and `::`-separated for an operation (`"package.component.worker::operation"`) — the same shape `OperationRequest`'s invocation id uses. A discovery id with an operation and an invocation id are the same string; `::` never appears anywhere except right before the operation.

`describe(?string $id = null)` is a single entry point over the rest of `ExplorerInterface`: it resolves an id to whichever of `getPackage()/getComponent()/getWorker()/getOperation()` matches, or, for `null` (there is no single “root” resource to look up), `{'summary' => ..., 'description' => ...}`,

```
'packages' => getPackages()};
```

```
$explorer->describe(); // {'summary' =>
..., 'description' => ..., 'packages' => getPackages()}
$explorer->describe('billing'); //
getPackage('billing')
$explorer->describe('billing.invoice'); //
getComponent(...)
$explorer->describe('billing.invoice.builder'); // getWorker(...)
$explorer->describe('billing.invoice.builder::createDraft'); //
getOperation(...)
```

`summary/description` are the package registry's own PHPDoc — kept alongside `packages` rather than replacing it, the same way every other level keeps its own `summary/description` alongside its children (`components/workers/operations`). `tree(null)` nests the same way, just with each package in `packages` deeply nested instead of the shallow `getPackage()` shape `describe(null)` holds.

More than 3 dot-separated segments before an operation, an empty segment, or an empty operation after `::` all throw `InvalidDiscoveryIdException` — a different exception from `InvalidOperationIdException` on purpose, even though the shape is identical once an operation is present: `describe()` also accepts a *partial* id (just a package, or a package and component) to browse with, something `OperationRequest::fromId()` (always a complete, 4-part invocation) never allows.

`tree(?string $id = null)` resolves the same id the same way, but nests every level's children instead of stopping at that one node — a worker comes back with its own operations nested right in:

```
$explorer->tree('billing.invoice.builder');
// [
//   'id' => 'billing.invoice.builder',
//   'name' => 'builder',
//   'summary' => 'Invoice Builder.',
//   'description' => '...',
//   'operations' => [
//     ['id' => 'billing.invoice.builder::build', 'name' => 'build', /* ... */],
//     ['id' => 'billing.invoice.builder::cancel', 'name' => 'cancel', /* ... */],
//   ],
// ]
```

A package nests its `components`, each of those its own `workers`, each of those its own `operations` — operations are always the leaves, nothing nests under them, so an operation id resolves exactly like `describe()`. The same policy-based pruning as the rest of Explorer applies at every level: a worker left with zero visible operations does not appear in its component's `workers`, and so on up to the root.

For `null`, same as `describe(null)`: `{'summary' => ..., 'description' => ..., 'packages' => [...]}`, except each package in `packages` is deeply nested here instead of the shallow `getPackage()` shape `describe(null)` holds:

```
$explorer->tree();
// [
//     'summary' => '...',          // the package registry's own PHPDoc, first sentence.
//     'description' => '...',      // the rest of the same PHPDoc.
//     'packages' => [
//         ['id' => 'billing', 'name' => 'billing', 'summary' => '...', 'description'
// => '...', 'components' => [/ * ... */]],
//     ],
// ]
```

Every other level in this tree is {id, name, summary, description, <children>}; the root is the same shape minus id/name (a registry is not a ServiceInterface with an identity, unlike a package/component/worker) — summary/description still have the same honest source as everywhere else, the registry class' own PHPDoc, so there is no reason for them to be missing just because id/name do not apply.

Operation ≠ Job

An “operation” here is simply a public method of a worker, discovered via reflection. It has nothing to do with Backbone's own formally-registered JobInterface/#[Job] concept — a worker's operation may use zero, one, or several real jobs internally, and the dispatcher neither knows nor cares.

Documenting an Operation: #[Operation]

derafu/backbone's #[Operation] attribute (see [Controlling Which Operations Can Be Dispatched](#) above) can also carry documentation reflection/PHPDoc cannot produce on its own:

```
use Derafu\Backbone\Attribute\Operation;

#[Operation(
    name: 'Create a draft invoice',          // overrides the PHPDoc summary, if given.
    description: 'Builds a draft from the given data, without emitting it.', //
    overrides the PHPDoc description.
    parameters: [
        'number' => ['example' => 'F-001'],
        'amount' => ['example' => 15000, 'description' => 'Amount in the smallest
currency unit.'],
    ],
    results: [
        'success' => ['description' => 'The created draft.', 'example' => ['id' =>
'DR-001']],
        MissingParameterException::class => ['description' => 'A required parameter
was not provided.'],
    ],
)]
public function build(string $number, int $amount): array
```

`Inspector::getPublicMethods()` merges whatever is given here on top of what reflection and PHPDoc already produced, before any policy or documentation consumer ever sees it — see [Operations](#) for the full attribute reference (every property, what each one means, and why it's not a service-defining attribute like `#[Job]/#[Handler]/#[Strategy]`).

Caching Reflection: `CachedInspector`

`getClassDoc()` and `getPublicMethods()` parse PHPDoc for every method of a class — the same result every time for a given class, until the next deploy. `CachedInspector` decorates any `InspectorInterface` with a PSR-6 `CacheItemPoolInterface`, caching exactly those two calls:

```
use Derafu\BackboneDispatcher\Service\Reflection\CachedInspector;
use Derafu\Cache\Adapter\PhpFilesCache;

$inspector = new CachedInspector(
    new Inspector(),
    new PhpFilesCache('backbone_dispatcher', '/var/cache/backbone_dispatcher'), // any
    PSR-6 Psr\Cache\CacheItemPoolInterface.
    ttl: 3600,
);
```

There is no default pool: `$cache` must always be given explicitly — `derafu/cache`'s `PhpFilesCache/FileSystemCache` (shown above), any other PSR-6 implementation (Redis, APCu, an in-memory one for tests), or `derafu/cache`'s `LocalCacheFactory` if the backend is itself a runtime choice. If caching isn't wanted at all, the simplest option is not using `CachedInspector` — inject the plain `Inspector` wherever `InspectorInterface` is expected instead, no flag needed anywhere.

Set `ttl: 0` to skip the pool on every call without swapping which `InspectorInterface` is injected — useful when the choice comes from runtime configuration rather than wiring. There is no invalidation logic inside the package on purpose: whether an entry outlives a deploy is entirely up to which pool gets injected — a file-based pool under the system temp directory is exactly as ephemeral as the process using it; a shared backend (Redis, APCu) is the caller's to flush, or not, as part of their own deploy.

`isOperation()`, `hasOperationAttribute()` and `getOperationParameters()` are deliberately **not** cached, in `CachedInspector` or otherwise: they never parse PHPDoc, they run on every single dispatch regardless of which tier or policy is used, and reading them through a cache backend could easily cost more than the reflection they would replace.

Never Failing While Exploring: `SafeExplorer`

`Explorer/ExplorerInterface` throw — `OperationNotFoundException`, `OperationNotAllowedException`, `InvalidDiscoveryIdException`, or any of `derafu/backbone`'s own

“not found” exceptions when a given package/component/worker doesn’t exist. `SafeExplorer` wraps any `ExplorerInterface` the same way `SafeDispatcher` wraps `TypedDispatcher`: no `Throwable` ever crosses back to the caller — every one of `ExplorerInterface`’s 10 public methods instead returns a `DiscoveryResultInterface`:

```
use Derafu\BackboneDispatcher\Service\Discovery\SafeExplorer;

$safeExplorer = new SafeExplorer($explorer, environment: 'prod', debug: false);

$result = $safeExplorer->tree('billing.invoice.builder::build');

if ($result->isSuccess()) {
    $operation = $result->getValue();
} else {
    $problem = $result->getProblem(); // Same RFC 7807-shaped ProblemDetail as
    SafeDispatcher's.
}
```

`DiscoveryResultInterface` has the same `isSuccess()/getValue()/getProblem()` shape as `OperationResultInterface`, but is a deliberately separate type: one is about dispatching an operation, the other about exploring the package tree, and keeping them independent means either one is free to grow its own data later without dragging the other along. It has no `getMetadata()`: `SafeExplorer` does not measure `ExecutionMetadata`.

Last updated on 10/10/2026