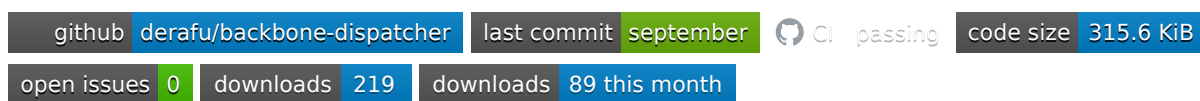

Docs » Base for Applications Category » Backbone Dispatcher Project

Backbone Dispatcher Project

Safely Invoke Any Backbone Operation, From Anywhere

Anonymous · 22/09/2026 · #php

Safely Invoke Any Backbone Operation, From Anywhere



Generic invocation and introspection for [Backbone](#) services: turn any worker's operation into something safely callable by id, with plain array parameters, from outside PHP's own type system.

Why

A [Backbone](#) worker's public methods are ordinary, statically-typed PHP methods — great for an in-process PHP caller, useless for anything else. Backbone Dispatcher adds a generic invocation layer on top: it turns a string operation id (`"package.component.worker::operation"`) plus a plain associative array of parameters into a real, reflection-resolved, type-coerced method call on the right worker, and turns whatever comes back — a return value **or** an uncaught exception — into a plain, serializable shape.

That is what makes it possible for a caller with no concept of PHP reflection, PHP's type system, or PHP exceptions — Python code across a [phpy](#) boundary, an HTTP client through [Backbone API](#), or anything else — to invoke any Backbone operation and get back a predictable, serializable answer either way.

Installation

```
composer require derafu/backbone-dispatcher
```

The Three-Tier Dispatcher

Each tier wraps the one below it and adds exactly one thing — none of them re-implement the others' job.

DirectDispatcherInterface

```
public function dispatch(
    string $package, string $component, string $worker,
    string $operation, array $params = []
): mixed;
```

Resolves the worker from the package registry, resolves and validates the parameters against the operation's real method signature, and calls it. Returns exactly what the operation returned, as a real PHP value — no wrapping. Any Throwable propagates unaltered. Use this directly only when you're a PHP-only caller happy to get real domain objects back and to handle exceptions yourself.

TypedDispatcherInterface

```
public function dispatch(OperationRequestInterface $request):
    OperationResultInterface;
```

Adapts `DirectDispatcher` to the `OperationRequest` → `OperationResult` shape. Still does **not** catch exceptions — a thrown `Throwable` propagates unaltered, and a successful call always has `isSuccess() === true` (there is no "failure" `OperationResult` coming out of this tier, only a thrown exception). It doesn't serialize the return value either: a domain object comes back as the real object.

SafeDispatcherInterface

```
public function dispatch(OperationRequestInterface $request):
    OperationResultInterface;
```

Same signature as `TypedDispatcher`, but **never throws**: any `Throwable` becomes a failure `OperationResult` carrying a `ProblemDetail`. It's also the only tier that **serializes** the success value before returning it — because it's the tier meant to actually cross a process/language boundary, where an uncaught PHP exception is a dead end.

```
$inspector = new Inspector();

$directDispatcher = new DirectDispatcher(
    $registry,
    $inspector,
    new Resolver(
        $inspector,
        new Caster(new ObjectFactoryRegistry(fallback: new FromArrayDeserializer())),
        new Validator(),
    ),
    // 4th argument, OperationPolicyInterface, defaults to AllowAllOperationPolicy –
    // see below.
);

$safeDispatcher = new SafeDispatcher(
```

```
new TypedDispatcher($directDispatcher),
new Serializer(),
environment: 'prod',
debug: false,
);
```

There is no bundled DI wiring for this chain — wire it however your application already wires services (a `config/services.yaml`, a PHP-DI container, etc.), using the composition above as the reference.

`DirectDispatcher` invokes the resolved operation with a native named-argument call (`$worker->$operation(...$args)`) rather than a generic invoker library: `Resolver` already produces `$args` as a plain, name-keyed, fully-cast array, so nothing is left for an invoker to resolve that hasn't already been resolved.

Controlling Which Operations Can Be Dispatched:

`OperationPolicyInterface`

By default, any public method of a worker is dispatchable — the historical behavior, kept as `AllowAllOperationPolicy`, `DirectDispatcher`'s 4th constructor argument's default value. **This default is convenient, not recommended:** “any public method” includes infrastructure methods a worker gets from `JobsAwareTrait`/`HandlersAwareTrait`/`OptionsAwareTrait` (`getJobs()`, `setOptions()`, ...) and from `ServiceInterface` itself (`getId()`, `getName()`, `getDescription()`) — none of it business logic, all of it just as dispatchable as a real operation while this policy is active. `TaggedOperationPolicy` is the recommended choice for anything reached from outside PHP: it only allows what is explicitly tagged `#[Operation]`, which is the one real signal for “this is meant to be exposed” — reflection alone cannot tell “a public method that happens to exist” apart from “an operation” (see `AllowAllOperationPolicy`'s own docblock). Two other policies ship with the package, and swapping the one wired into `DirectDispatcher` is the only change needed — nothing else in the three-tier chain knows a policy exists:

```
use Derafu\BackboneDispatcher\Service\Policy\TaggedOperationPolicy;
use Derafu\BackboneDispatcher\Service\Policy\AllowListOperationPolicy;

// Only methods tagged with derafu/backbone's #[Operation] attribute.
$policy = new TaggedOperationPolicy($registry, $inspector);

// Only operations matching one of these ids. fnmatch() wildcards allowed,
// e.g. "billing.invoice.builder:*" for every operation of that worker,
// or "billing.*" for an entire package.
$policy = new AllowListOperationPolicy([
    'billing.invoice.builder::build',
    'billing.invoice.builder::cancel',
]);

$directDispatcher = new DirectDispatcher(
    $registry,
```

```
$inspector,  
$resolver,  
$policy,  
);
```

Two guards run in `DirectDispatcher::dispatch()`, in this order, before an operation is ever resolved or invoked — so every tier built on top of it enforces both without knowing they exist:

- Does the operation exist at all, as a public method declared on the worker? Independent of which policy is configured — throws `OperationNotFoundException` otherwise.
- Does the active `OperationPolicyInterface` allow it? Throws `OperationNotAllowedException` otherwise.

Explorer accepts the same `OperationPolicyInterface` as an optional constructor argument, so documentation never advertises an operation the dispatcher would then reject — without one, it lists everything, matching `AllowAllOperationPolicy`. With one, the pruning goes all the way up the tree: `getOperations()` drops operations it rejects, and `getWorkers()/getComponents()/getPackages()` drop any branch left with zero visible operations underneath — a worker nobody may call does not show up at all, not even as an empty entry.

Implementing `OperationPolicyInterface` yourself (a single `isAllowed()` method) covers any other rule — e.g. combining policies, or checking against the current user's permissions.

Dispatching an Operation

```
use Derafu\BackboneDispatcher\ValueObject\OperationRequest;  
  
$request = OperationRequest::fromId(  
    'billing.invoice.builder::build',  
    ['number' => 'F-001', 'amount' => 15000],  
);  
  
$result = $safeDispatcher->dispatch($request);  
  
if ($result->isSuccess()) {  
    $invoice = $result->getValue(); // already serialized: array/scalar, never a raw  
    PHP object.  
} else {  
    $problem = $result->getProblem(); // RFC 7807-shaped ProblemDetail.  
}
```

`OperationRequest::fromId()` parses `"package.component.worker::operation"` — an id with the wrong shape (missing `::`, wrong number of `.`-segments, any empty piece) throws `InvalidOperationIdException`. `::` (not a single `:`) on purpose: `derafu/backbone` already uses a single `:` for its own `.job:name/.handler:name/.strategy:name` ids — this identifier is a dispatcher-only concept, unrelated to that family, and `::` keeps it visually distinct.

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.

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`.

Exceptions

`InvalidOperationIdException` — malformed "package.component.worker::operation" id.
`InvalidDiscoveryIdException` — malformed discovery id passed to `Explorer::describe()`.

`ResolverException`

└─ `InvalidParameterTypeException` — a scalar parameter has the wrong native type.
 └─ `MissingParameterException` — a required parameter is absent.

`ObjectFactoryException`

└─ `ClassNotFoundException` — `fromArray()` target class doesn't exist.

- └─ FromArrayMethodNotFoundException – target class has no static fromArray().
- └─ NoDeserializerFoundException – no registered deserializer, and the fallback failed too.

OperationNotFoundException – the operation does not exist as a public method of the worker.

OperationNotAllowedException – the operation exists, but the active OperationPolicyInterface rejects it.

Every exception exposes a semantic static factory (`InvalidOperationIdException::forId()`, `MissingParameterException::forParameter()`, etc.) instead of a public constructor, and is translatable via [derafu/translation](#).

Requirements

PHP 8.5+. Depends on [derafu/backbone](#), [derafu/cache](#) and [derafu/translation](#).

Last updated on 22/09/2026