

Docs » Base for Applications Category » Backbone API Project

Backbone API Project

HTTP API With Zero Per-Operation Controllers

Anonymous · 22/09/2026 · #php

HTTP API With Zero Per-Operation Controllers

github	derafu/backbone-api	last commit	august	CI	passing	code size	76.9 KiB	open issues	0
downloads	91	downloads	36 this month						

Turns every operation of every worker registered in a [Backbone](#) package registry into an HTTP endpoint automatically — no per-operation controller code, self-documenting as HATEOAS resources and an OpenAPI 3.1 spec.

Why

Instead of writing `POST /invoices/build → InvoiceController::build()` by hand for each operation, you write one generic route pointing at one controller, and the URL path itself tells the package which worker method to call. The actual resolution and invocation is delegated entirely to [derafu/backbone-dispatcher](#), which is transport-agnostic — this package only deals with the HTTP-specific concerns: parsing the route, listing packages/components/workers as browsable resources, serving OpenAPI documentation, and extracting parameters from the request body.

Installation

```
composer require derafu/backbone-api
```

There is no bundled route table or DI wiring — wiring [Dispatcher](#) and a controller into your framework's routes and container is left to your application.

Routing Convention

`Router::parse()` resolves an incoming request into up to four path segments:

```
/[api]:package/:component/:worker/:operation
```

The `/api` prefix is optional and stripped if present — `/api/billing/document` and `/billing/document` resolve identically. A path with more than 4 segments throws `InvalidRouteException`. Trailing

segments are simply omitted rather than required, which is what drives the dispatch behavior below.

Two special one-segment paths are intercepted before being treated as a package name:

- Empty path or `index` → the root HATEOAS listing.
- `openapi-docs.json` → the generated OpenAPI document.

```
public function dispatch(ServerRequestInterface $request): mixed
{
    $route = $this->router->parse($request);

    return match (true) {
        $route->getId() === 'index' || $route->getId() === null =>
        $this->handleRoot(),
        $route->getId() === 'openapi-docs.json' => $this->documenter->document(),
        $route->getComponent() === null => $this->handlePackage($route->getPackage()),
        $route->getWorker() === null => $this->handleComponent($route->getPackage(),
        $route->getComponent()),
        $route->getOperation() === null => $this->handleWorker(...),
        default => $this->handleOperation($request, ...),
    };
}
```

So `GET /api/billing` lists `billing`'s components, `GET /api/billing/document` lists that component's workers, and only a full 4-segment path actually invokes an operation.

Invoking an Operation

Every operation is invoked with the parameters read from the JSON request body's `parameters` key — nothing else (no query string, no path params beyond routing):

```
POST /api/billing/invoice/builder/build
```

```
Content-Type: application/json
```

```
Accept: application/json
```

```
{ "parameters": { "number": "F-001", "amount": 15000 } }
```

```
$requestContent = json_decode($request->getBody()->getContents(), true);
```

```
$params = $requestContent['parameters'] ?? [];
```

```
$operationRequest = new OperationRequest($package, $component, $worker, $operation,
$params);
```

```
return $this->dispatcher->dispatch($operationRequest); // SafeDispatcherInterface —
never throws.
```

`handleOperation()` returns the `OperationResultInterface` itself, unwrapped — `AbstractController` (below) is the single place that turns it into the actual response, since it is also the place that already builds every other response shape this package produces.

Response Shape

A successful, JSON-wanting request (Accept: `application/json` or `*/*`) gets wrapped in a small envelope — for an operation result, `meta.timestamp/meta.data_type` come straight from `OperationResultInterface::getMetadata()->getTimestamp()/::getDataType()` (see [Backbone Dispatcher](#)), not recomputed from the already-serialized value:

```
{
  "meta": { "timestamp": 1755500000.123456, "data_type": "integer" },
  "data": 12
}
```

Two results bypass the envelope entirely: a PSR-7 `ResponseInterface` returned by the underlying dispatch (passed through unchanged), and the OpenAPI document itself (detected by its own `openapi` key). If the request doesn't ask for JSON, the raw result is returned unwrapped for the framework layer to render however it sees fit.

Failure: a Real Status Code, Never a Silent 200

`SafeDispatcherInterface` never throws, so `AbstractController` is also the one place that decides what a failed operation actually looks like to the client — a real PSR-7 response (built via an injected `Psr\Http\Message\ResponseFactoryInterface`, so this package stays implementation-agnostic about which PSR-17 factory produces it) carrying:

- The resolved HTTP status, from `HttpStatusResolver` (below) — never a default 200 masking a real failure.
- The exact same `ProblemDetailInterface::toArray()` body [Backbone Dispatcher](#) and [Backbone Console](#) already produce — the same failure looks the same across every transport.

```
$problem = $result->getProblem();
$status = $this->httpStatusResolver->resolve($problem->getThrowable()->getClass());

$response = $this->responseFactory
    ->createResponse($status)
    ->withHeader('Content-Type', 'application/json')
;
$response->getBody()->write((string) json_encode($problem->toArray()));
```

HttpStatusResolver: Mapping Exceptions to Status Codes

Shared by two consumers: `AbstractController` above (a real failure’s actual status) and `Documenter` below (what to document for each `#[Operation(results: ...)]` scenario) — the same resolution, kept in one place instead of two copies that could drift apart. Only `derafu/backbone-dispatcher`’s own 7 generic exceptions are mapped (the same 7 [Backbone Console](#)’s `DefaultExitCodeResolver` maps, mirroring that same split) — anything else, including a business-specific exception, falls back to 500:

Status	Exception
200	('success')
403	<code>OperationNotAllowedException</code>
422	<code>MissingParameterException</code> , <code>InvalidParameterTypeException</code> , <code>NoDeserializerFoundException</code> , <code>ClassNotFoundException</code> , <code>FromArrayMethodNotFoundException</code>
500	anything else, including <code>OperationNotFoundException</code> and any business-specific exception

There is no built-in way to extend this mapping with a project’s own business exceptions yet — unlike `DefaultExitCodeResolver`, `HttpStatusResolver` is not currently designed to be subclassed for that. A project needing its own status codes today would wrap or replace it in its own DI wiring.

Autodiscovery

One shared criterion for what counts as “visible”, applied consistently across both surfaces — because both are walked from the exact same source:

Explorer composes over `derafu/backbone-dispatcher`’s own `ExplorerInterface` rather than reimplementing its walk, only adding HATEOAS `_links` on top of whatever it returns — `GET /api/billing/document/builder` returns the worker’s `_links` plus its operations. Every id, name, description and policy-based visibility rule (see [Backbone Dispatcher](#)) comes straight from the delegate: with a policy wired into it, a worker nobody may call does not show up here either.

Documenter generates the OpenAPI 3.1 document served at `GET /api/openapi-docs.json`, walked from `Explorer::tree()` — the exact same nested, policy-pruned structure `Explorer` itself is built on, not a separate traversal of the package registry. This used to not be true: `Documenter` had its own, independent notion of “visible” (any method tagged with Backbone’s `#[Operation]` attribute, optionally narrowed by a *second*, separately-injected policy instance), which could silently drift from what a real `DirectDispatcher` would actually accept — an operation the active `OperationPolicyInterface` allowed, but nobody had gotten around to tagging, was dispatchable and browsable yet absent from the spec, understating the real attack surface. There is no way to configure `Documenter` differently from `Explorer` anymore: whatever `OperationPolicyInterface` was wired into the `ExplorerInterface` `Explorer` composes over is the one and only thing that decides what gets documented.

`#[Operation]` still exists and is still useful — for overriding a parameter’s reflected type/description/example, or the operation’s own name/description, when the reflected PHPDoc isn’t enough (see [Backbone Dispatcher](#)) — it just no longer gates *whether* something gets documented, only how it looks once it is. If you want a real, non-`#[Operation]`-tagged public method to disappear from both `Explorer` and `Documenter` at once, that is exactly what [TaggedOperationPolicy](#) is for — wired once, where the dispatch chain itself is built, never in `backbone-api`.

Every documented operation is generated as a single OpenAPI `post` entry (there’s no GET/PUT/DELETE distinction), and its request/response schema is built from the reflected parameter types, following the same type-name vocabulary as `backbone-dispatcher`’s `Caster::resolveType()` (`string`, `number`, `integer`, `boolean`, `array`, `object`).

Operation ≠ Job

Same terminology note as [Backbone Dispatcher](#): an “operation” here is any public method found via reflection, unrelated to Backbone’s own `JobInterface/#[Job]` concept.

Requirements

PHP 8.5+. Depends on [derafu/backbone](#), [derafu/backbone-dispatcher](#), `psr/http-message`, and `psr/http-factory` (for building the real failure response — a concrete PSR-17 implementation, e.g. `nyholm/psr7`, is the consuming application’s to provide).

Last updated on 22/09/2026