

[Docs](#) » [Base for Applications Category](#) » [Backbone Console Project](#)

Backbone Console Project

Turn Any Backbone Operation Into a CLI Command

Anonymous · 08/10/2026 · #php

Turn Any Backbone Operation Into a CLI Command

github derafu/backbone-console last commit last tuesday  CI passing code size 96.7 KiB open issues 0

downloads 22 downloads 6 this month

Exposes every operation a [SafeExplorerInterface](#) discovers as its own auto-discovered [Symfony Console](#) command — no per-operation Command subclass, ever.

Why

[Backbone Dispatcher](#) already turns a string operation id and a plain array of parameters into a safe, serializable call. This package is one more transport on top of it — the same idea [Backbone API](#) is for HTTP, but for a standalone CLI process, meant to be run by hand or invoked from any other language via `exec()/system()/subprocess.run()`, not just from PHP.

A business library can expose hundreds of operations. `OperationCommandLoader` builds a `Symfony\Component\Console\Command\Command` for exactly one, on demand, only when a `bin/console` invocation actually needs it — never all of them upfront.

Installation

```
composer require derafu/backbone-console
```

Usage

Compose a Kernel with [derafu/console](#)'s `ConsoleKernelTrait`, and install `OperationCommandLoader` as the `CommandLoaderInterface`:

```
use Derafu\BackboneConsole\Service\OperationCommandLoader;
use Derafu\BackboneDispatcher\Contract\SafeDispatcherInterface;
use Derafu\BackboneDispatcher\Contract\SafeExplorerInterface;
use Derafu\Console\ConsoleKernelTrait;
```

```

use Symfony\Component\Console\Application as SymfonyConsoleApplication;
use Symfony\Component\DependencyInjection\ContainerInterface;

class ConsoleApplication extends YourOwnKernel // e.g. your business library's own
Kernel.
{
    use ConsoleKernelTrait {
        createConsoleApplication as private buildBaseConsoleApplication;
    }

    protected function createConsoleApplication(
        ContainerInterface $container
    ): SymfonyConsoleApplication {
        $application = $this->buildBaseConsoleApplication($container);

        $application->setCommandLoader(new OperationCommandLoader(
            $container->get(SafeExplorerInterface::class),
            $container->get(SafeDispatcherInterface::class),
        ));

        return $application;
    }
}

```

```

// bin/console
use App\ConsoleApplication;
use Derafu\Console\Runtime;

require dirname(__DIR__) . '/vendor/autoload.php';

exit(Runtime::run(fn (array $context): ConsoleApplication => new ConsoleApplication(
    $context['APP_ENV'],
    (bool) $context['APP_DEBUG'],
)));

```

Extending your own Kernel directly (rather than `Derafu\Console\Kernel`) is deliberate whenever the console needs the exact same `services.yaml`-wired services (like `SafeExplorerInterface`/`SafeDispatcherInterface` above) as the rest of the project — see [Console](#).

Every discovered operation becomes its own command, named by converting its id's `./::` separators to Symfony Console's own `:` convention:

```
billing.invoice.builder::build → billing:invoice:builder:build
```

```
bin/console list
bin/console help billing:invoice:builder:build
```

The Request: One File (or STDIN), Any Format

A `Worker`'s operations can each take arbitrarily different parameters — there is no fixed set of CLI flags to define per command, only reflection (via `SafeExplorerInterface`) knows the shape at runtime. So the request is always a single JSON, YAML, or XML document, with the operation's parameters under a `"parameters"` key — the same shape an HTTP body would use:

```
echo '{"parameters": {"number": "F-001", "amount": 15000}}' \
  | bin/console billing:invoice:builder:build

bin/console billing:invoice:builder:build request.json
bin/console billing:invoice:builder:build request.yaml
bin/console billing:invoice:builder:build - # "-": explicit STDIN.
```

The `input` argument is optional — omitted, or given as `-`, reads from STDIN. Format is auto-detected from content: a leading `<` is tried as XML, then JSON, then YAML (YAML is a syntactic superset of JSON, so trying it first would mean JSON is never actually detected as its own format).

Content starting with `{/[` is treated as an unambiguous signal that JSON was intended: if it fails to parse as JSON, that is a real, thrown error (`EX_DATAERR`, see below) rather than a silent fallback to YAML. Without this, some malformed JSON parses successfully as something else instead of failing loudly — a trailing comma (`{"a": 5, }`) is invalid JSON but valid YAML flow-style, so it used to silently reinterpret into a document simply missing whatever came after the comma. The trade-off: a hand-written top-level YAML *flow-style* document (`{a: 5}` written as YAML on purpose, not as a JSON typo) is no longer accepted — YAML's more common block style (`a: 5` on its own line) is completely unaffected.

The Response: Same Format as the Request, or Forced With `--output`

Without `--output`, the response is written to STDOUT in the same format the request came in as — a successful one is `{"meta": {"timestamp": ..., "data_type": ...}, "data": <value>}` (the same envelope [Backbone API](#) uses, so a caller does not have to special-case which transport it is talking to); a failed one is `ProblemDetailInterface::toArray()` (see [Backbone Dispatcher](#), `extensions.timestamp/extensions.data_type` included there too), written to STDERR instead, alongside a resolved exit code (see below).

`--output <path>` and `--error-output <path>` write to a real file instead of STDOUT/STDERR, independently of each other — `--output` only ever applies on success, `--error-output` only ever on failure (a single invocation only ever hits one of the two branches, so they can even point at the same path without conflict). Piping with shell redirection (`>/2>`) already works, but conflates the structured payload with anything else the process might write; these options guarantee only the payload lands in the file:

```
bin/console billing:invoice:builder:build request.json --output=result.yaml
bin/console billing:invoice:builder:build request.json --error-output=problem.xml
```

The destination's own file extension (`.json/.yaml/.yml/.xml`) decides the *response* format, independent of the *request's* — an unrecognized or missing extension (e.g. `.txt`) falls back to the request's format, same as omitting the option entirely. `-` is accepted explicitly for `"STDOUT"/"STDERR"`, the same convention the `input` argument already uses.

A write failure is never silent: a missing destination directory or a permissions error is reported on the real `STDERR` and returns `GenericOperationCommand::EX_CANTCREAT` — never a false "success" while nothing was actually written.

Execution Metadata: `-v`

The response payload's *shape* never depends on where it is written, only on `-v/--verbose` — and only for how *much* ends up in it, never whether `meta/data_type/timestamp` are there at all (they always are). Without `-v`, `meta` has just `timestamp/data_type`, as shown above. With it, the rest of `ExecutionMetadata` (`startedAt`, `timing`, `memory`, `CPU`, `load average`) is merged into that same `meta` on success, or into `extensions` on failure (next to the already-present `debug/context/throwable`) — never a new top-level key:

```
bin/console -v billing:invoice:builder:build request.json
```

```
{
  "meta": {
    "timestamp": 1755500000.123456,
    "data_type": "App\\Entity\\Invoice",
    "startedAt": "2026-01-20T10:00:00+00:00",
    "finishedAt": "2026-01-20T10:00:00+00:00",
    "realTime": 0.0234,
    "userTime": 0.0198,
    "systemTime": 0.0012,
    "memoryUsed": 131072,
    "peakMemory": 4194304,
    "pid": 12345,
    "loadAverage1Min": 0.52,
    "loadAverage5Min": 0.61,
    "loadAverage15Min": 0.58
  },
  "data": { "id": "INV-001" }
}
```

Exit Codes

`bin/console help <command>` always lists every exit code that specific command can return — the fixed ones below, plus whatever the injected `ExitCodeResolverInterface` reports via `describe()` (see next section).

Code	Meaning
0	Success.
1	The operation failed, and nothing more specific applies (<code>DefaultExitCodeResolver</code> 's fallback).
10–16	One of derafu/backbone-dispatcher's own 7 generic exceptions — see below.
65 (EX_DATAERR)	The request could not be parsed as JSON/YAML/XML.
66 (EX_NOINPUT)	The input file does not exist or is not readable.
70 (EX_SOFTWARE)	An unexpected internal error — a bug, not a usage or business problem.
73 (EX_CANTCREAT)	<code>--output/--error-output</code> could not be created.

65/66/70/73 are `sysexits(3)` codes — a real BSD convention (`/usr/include/sysexits.h` on macOS/BSD), chosen because they start at 64 specifically to avoid colliding with whatever small integers other programs already use for their own exit codes. 2 (Symfony Console's own `Command::INVALID`) is reserved but currently unused by this package.

ExitCodeResolverInterface: Mapping Exceptions to Codes

Every failure not caused by this command itself (a `ProblemDetailInterface`, produced by the dispatch) goes through `ExitCodeResolverInterface::resolve()`. `DefaultExitCodeResolver` — the default — already maps the 7 exceptions generic to *any* Backbone-based project, since they mean the same thing regardless of domain:

Code	Exception
10	<code>OperationNotFoundException</code>
11	<code>OperationNotAllowedException</code>
12	<code>MissingParameterException</code>
13	<code>InvalidParameterTypeException</code>
14	<code>ClassNotFoundException</code>
15	<code>FromArrayMethodNotFoundException</code>
16	<code>NoDeserializerFoundException</code>

A project that also wants its *own* business exceptions mapped extends `DefaultExitCodeResolver` rather than starting from nothing, layering its mapping on top via `parent::`:

```
use Derafu\BackboneConsole\Service\DefaultExitCodeResolver;
use Derafu\BackboneDispatcher\Contract\ProblemDetailInterface;

class MyExitCodeResolver extends DefaultExitCodeResolver
{
    public function resolve(ProblemDetailInterface $problem): int
    {
        return match ($problem->getThrowable()->getClass()) {
            MyBusinessException::class => 20,
            default => parent::resolve($problem),
        };
    }

    public function describe(): array
    {
        return [MyBusinessException::class => 20] + parent::describe();
    }
}
```

Codes `>= 10` are the convention for a project's own mapping — clear of Symfony Console's `0/1/2`, `DefaultExitCodeResolver`'s own `10-16`, and the `64-78` `sysexits(3)` range this package uses internally.

Requirements

PHP 8.5+. Depends on [derafu/backbone-dispatcher](#), [derafu/console](#), `derafu/xml`, and `symfony/console/symfony/yaml`.

Last updated on 08/10/2026