

Usage

Usage

Anonymous · 10/10/2026

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.

Last updated on 10/10/2026