

The Three-Tier Dispatcher

The Three-Tier Dispatcher

Anonymous · 10/10/2026

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.

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.

Last updated on 10/10/2026