

Middleware

Middleware

Anonymous · 08/10/2026

Middleware

Derafu\Http processes every request through a chain of [PSR-15](#) middlewares. The chain is a plain list in your services file: the order of the list is the order of execution.

The Middleware Stack

```
Psr\Http\Server\RequestHandlerInterface:
  class: Derafu\Http\Service\RequestHandler
  public: true
  arguments:
    $middlewares:
      - '@Derafu\Http\Middleware\StaticFilesMiddleware'
      - '@Derafu\Http\Middleware\RequestFactoryMiddleware'
      - '@Derafu\Http\Middleware\ClientIpMiddleware'
      - '@Derafu\Http\Middleware RouterMiddleware'
      - '@Derafu\Http\Middleware DispatcherMiddleware'
      - '@Derafu\Http\Middleware\ResponseNormalizerMiddleware'

# Register each middleware as a service.
Derafu\Http\Middleware\StaticFilesMiddleware:
  arguments:
    $directory: '%kernel.project_dir%/public/static'
Derafu\Http\Middleware\RequestFactoryMiddleware: ~
Derafu\Http\Middleware\ClientIpMiddleware: ~
Derafu\Http\Middleware RouterMiddleware: ~
Derafu\Http\Middleware DispatcherMiddleware: ~
Derafu\Http\Middleware\ResponseNormalizerMiddleware: ~
```

`RequestHandler` walks the list one middleware at a time. Each middleware receives the request and the handler, and either returns a response itself (short-circuiting the chain) or calls `$handler->handle($request)` to continue.

One `RequestHandler` per request

Any exception thrown while the chain runs is caught by the `RequestHandler` and turned into a response. See [Error Handling](#).

Core Middlewares

These four work together and are required, in this order:

Order	Middleware	What it does
1	<code>RequestFactoryMiddleware</code>	Converts the PSR-7 request into a <code>Derafu\Http\Request</code> . Adds the <code>derafu.context</code> attribute with the kernel context, when there is one. Every middleware after it receives a <code>Derafu\Http\Request</code> .
2	<code>RouterMiddleware</code>	Gives the request the canonical form of its path (<code>/api//index</code> and <code>/api/%69ndex</code> are <code>/api/index</code>), builds the request context, matches the path against the routes and stores the match in the <code>derafu.route</code> attribute. Throws <code>RouteNotFoundException</code> (a 404) when no route matches, and <code>InvalidPathException</code> (a 400) when the path has no safe form (<code>..</code> , an escaped slash, a control character).
3	<code>DispatcherMiddleware</code>	Runs the handler of the matched route and stores whatever it returns in the <code>derafu.response</code> attribute, then continues the chain.
4	<code>ResponseNormalizerMiddleware</code>	Turns the value returned by the handler into a PSR-7 response, as described below.

The path that the rest of the pipeline sees

`RouterMiddleware` replaces the path of the request with its canonical form (see [Url](#)) before it goes on. Every middleware after it, and the handler of the route, sees the same path whatever way the client wrote it: a rule that decides by the text of the path (the protected paths of `derafu/auth`) and the router that reads its meaning are about the same path. Without it, `/api//index` is served by the route of `/api/index` and is not under a rule that protects `/api/index`. The query and the rest of the URI are not changed. A path that has no safe form is refused with a 400 before anything else runs.

How the response is normalized

The handler of a route can return a `Response`, any PSR-7 response, or plain data (an array, a string, an object).

- A `Derafu\Http\Response` is returned as is. If it has no `Content-Type`, the preferred content type of the request is added.
- Any other PSR-7 response is returned as is.
- Anything else is converted with the content type the client prefers (see [Requests and Responses](#)). For JSON it is encoded with `json_encode()`.
- If JSON encoding fails and the value is a string, it is sent as plain text instead. This is the case of raw binary content, like a PDF. If it is not a string, a `ResponseSerializationException` is

thrown.

The normalizer takes the value from the `derafu.response` attribute. That is why it has to go **after** `DispatcherMiddleware`. If that attribute is missing (the dispatcher did not run before it), the normalizer asks the next handler for the response instead, so it also works in a chain where it is not the last one. When nothing produces a response, the request fails with a 500 “No response was generated by the middleware chain”.

It also removes the `X-Powered-By` header.

Optional Middlewares

Static Files

`StaticFilesMiddleware` serves static assets directly. It should go first in the stack, so assets are served before any other middleware runs:

```
Derafu\Http\Middleware\StaticFilesMiddleware:
arguments:
  $directory: '%kernel.project_dir%/public/static'
  $cacheMaxAge: 86400 # Optional, in seconds. Default: 24 hours.
```

What it serves, and what it never does:

- Only GET and HEAD requests, and only for files whose extension is known to `ContentType` (`.css`, `.js`, `.png`, `.json`, `.txt`, etc.). Anything else goes on to the next middleware, normally ending in a 404.
- Only files inside `$directory`. The directory is resolved once with `realpath()`, so a symlinked directory (like a `current` release) works. A file reached through a symlink that points outside of it is not served.
- Never a path with `..`, a null byte, or a segment that starts with a dot (`.env`, `.git/`, `.htaccess`, etc.).
- The only exception is `.well-known` ([RFC 8615](#)) as the **first** segment, for example `/.well-known/security.txt`. Everything after it is checked as usual, so `/.well-known/.hidden.txt` is still rejected.

It adds `Cache-Control: public, max-age=...` and an `ETag`, and answers `304 Not Modified` when the client sends a matching `If-None-Match`. A `HEAD` request gets the same headers and no body.

Files without extension or with an unknown type

Some `.well-known` files have no extension (`apple-app-site-association`, `acme-challenge/<token>`) or a type `ContentType` does not know. The middleware does not handle them, even if they exist in `$directory`: it is a deliberate limit, because it cannot guess a safe `Content-Type`. Resolve them in one of these ways:

1. **In the web server (recommended for static content).** Serve the path before PHP gets involved, for example in Caddy:

```
handle /.well-known/* {
    root * /path/to/well-known
    file_server
}
```

2. **With a route that sets the type explicitly**, when the content is generated or must be controlled by the app:

```
well_known_aasa:
    path: /.well-known/apple-app-site-association
    handler: App\Controller\WellKnownController::appleAppSiteAssociation
```

```
public function appleAppSiteAssociation(): Response
{
    return (new Response())->asText($json, ContentType::JSON);
}
```

3. **Adding the type to ContentType**, if it is a file type the library should know in general. Then it is served like any other static file.

Certificates issued with ACME (such as Caddy's automatic HTTPS) already answer `acme-challenge` by themselves: do not route it through the application.

CORS

`CorsMiddleware` implements Cross-Origin Resource Sharing with path-based rules. Place it **first** in the stack, so preflight requests never reach the router or the dispatcher.

```
Derafu\Http\Middleware\CorsMiddleware:
    arguments:
        $rules:
            - path: '^/api'
              allowedOrigins: ['https://app.example.com']
              allowedMethods: ['GET', 'POST', 'OPTIONS']
              allowedHeaders: ['Content-Type', 'Authorization']
              allowCredentials: true
              maxAge: 3600
            - path: '^/webhook'
              allowedOrigins: ['*']
              allowedMethods: ['POST', 'OPTIONS']
              allowedHeaders: ['Content-Type']
```

Each rule has these keys. Only `path` is required; the others use permissive defaults:

Key	Default	Meaning
<code>path</code>	(required)	Regular expression matched against the request path
<code>allowedOrigins</code>	<code>['*']</code>	Allowed origins
<code>allowedMethods</code>	<code>GET, POST, PUT, PATCH, DELETE, OPTIONS</code>	Allowed methods
<code>allowedHeaders</code>	<code>Content-Type, Authorization, Accept</code>	Allowed request headers
<code>allowCredentials</code>	<code>false</code>	Whether credentials are allowed
<code>maxAge</code>	<code>3600</code>	How long, in seconds, a preflight answer can be cached

Rules are evaluated in order and the first match wins. With no rules at all, one rule matching every path with the defaults is used.

The request passes through untouched (no CORS headers) when it has no `Origin` header, when no rule matches its path, or when its origin is not allowed by the matching rule. Otherwise:

- An `OPTIONS` preflight request is answered right away with `204` and the CORS headers, plus `Access-Control-Max-Age`.
- Any other request continues down the chain, and the CORS headers are added to its response.
- With `allowedOrigins: ['*']` and `allowCredentials: true`, the origin of the request is reflected instead of `*`, as the specification requires.

The defaults are permissive

Client IP

`ClientIpMiddleware` decides **who the client of a request is**, once, and leaves it in the request for everything that comes after: the limit of requests, the limit of failed logins of [derafu/auth](#), the logs. Put it after `RequestFactoryMiddleware` and before `ThrottleMiddleware` and the router.

The address of the connection (`REMOTE_ADDR`) is the one of whoever connected: the client, or the proxy that is in front of the application. The headers that proxies add (`X-Forwarded-For`, `CF-Connecting-IP`...) are written by whoever sends the request, so **a client can send them with any address**. They are only to be believed when they come from a proxy that the application trusts, and which proxies those are depends on where the application runs, so it is configured, not guessed:

- **Without trusted proxies** (the default) the client is the address of the connection, and **no header is read**. It is the right answer when the application is the one that receives the connections.

- **With trusted proxies** the headers are read only if the connection comes from one of them. A connection from any other address is the client, whatever its headers say.

Variable	Default	What it is
HTTP_TRUSTED_PROXIES	none	The proxies that are trusted, separated by commas: addresses (198.51.100.1), ranges in CIDR notation (198.51.100.0/24, 2001:db8::/32) and the shortcuts loopback, private and link-local.
HTTP_CLIENT_IP_HEADERS	X-Forwarded-For	The headers with the address of the client, separated by commas, in order of preference.
HTTP_CLIENT_NETWORK_IPV4_PREFIX	32	The bits of the network of an IPv4 client (see below).
HTTP_CLIENT_NETWORK_IPV6_PREFIX	64	The bits of the network of an IPv6 client.

The shortcut `private` is the networks that are not routed on the Internet (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 and fc00::/7), which are the ones of a network of containers. The ranges that a service publishes (the ones of a CDN, for example) are written as they are: they are not in the code because they change. A trusted proxy, a header or a prefix that is not valid is an error when the middleware is created, not something that is ignored.

```
# The application is behind a proxy of the network of containers, that says who
# the client is in X-Forwarded-For.
HTTP_TRUSTED_PROXIES=private

# Behind a CDN: only its ranges are believed, and the header that it overwrites.
HTTP_TRUSTED_PROXIES=203.0.113.0/24,2001:db8::/32
HTTP_CLIENT_IP_HEADERS=CF-Connecting-IP
```

How the client is found

Each header is read as the list of the addresses that the request went through, from the client to the proxy that is closest to the application. A header that has only one address (CF-Connecting-IP) is a list of one, and Forwarded ([RFC 7239](#)) is read by its `for` parameters. The first header, in the order of `HTTP_CLIENT_IP_HEADERS`, that gives an address wins.

The list is read **from the end**: the client is the first address that is not a trusted proxy. The first address of a list is the one that the client wrote, so it is not to be believed when there is a proxy between: each proxy adds at the end what it saw. With `X-Forwarded-For: 198.51.100.7, 203.0.113.9` and a connection from a trusted proxy, the client is `203.0.113.9`. If all the addresses are trusted proxies, the client is the first one. What is not an address (`unknown`, `_hidden`) is skipped, and ports, brackets and quotes are accepted (`[2001:db8::1]:443`).

What it leaves in the request

Attribute	Value
client_ip	The address of the client, normalized (see Ip), or unknown if the connection has none.
client_network	The network of the client, in CIDR notation: the address with the bits of its host set to zero, and the prefix (203.0.113.77/32, 2001:db8:1:2::/64). unknown if the connection has no address.

The network is what a limit should count by. A client of IPv6 has a whole /64 (or more), and changing its address within it costs nothing, so counting by address does not limit anything. By default the network is the /32 for IPv4 (the address itself, a network of one) and the /64 for IPv6. The value is used as a key: a limit counts by the whole text, so 203.0.113.77/32 and 203.0.113.77/24 are not the same client.

```
use Derafu\Http\Middleware\ClientIpMiddleware;

$ip = $request->getAttribute(ClientIpMiddleware::ATTRIBUTE);           //
"203.0.113.9"
$network = $request->getAttribute(ClientIpMiddleware::NETWORK_ATTRIBUTE); //
"2001:db8:1:2::/64"

// Without the middleware in the pipeline, these give the address of the
// connection (never a header).
ClientIpMiddleware::ipOf($request);
ClientIpMiddleware::networkOf($request);
```

Throttle

`ThrottleMiddleware` limits the number of requests per client with the [Symfony Rate Limiter](#), which you have to install:

```
composer require symfony/rate-limiter
```

The limit is configured on the `RateLimiterFactory`:

```
App\Middleware\ThrottleMiddleware: ~

Symfony\Component\RateLimiter\RateLimiterFactory:
  arguments:
    $config:
      id: 'throttle'
      policy: 'fixed_window'
      limit: 50
      interval: '24 hour'
    $storage: '@Symfony\Component\RateLimiter\Storage\StorageInterface'
```

```
Symfony\Component\RateLimiter\Storage\StorageInterface:
    class: Symfony\Component\RateLimiter\Storage\CacheStorage
```

Place it after `ClientIpMiddleware` and before `RouterMiddleware`, so the limit is checked before any route is dispatched.

Accepted requests get the `X-RateLimit-Limit` and `X-RateLimit-Remaining` headers. When the limit is exceeded, the request is rejected with a `TooManyRequestsException` (HTTP 429) that carries `Retry-After`, `X-RateLimit-Reset` and the two headers above. See [Error Handling](#).

By default it limits every request, with one counter per **network of the client** (the `client_network` of [Client IP](#)). The headers of the request are never read by this middleware: they are the client's, so counting by them would let it choose its own counter and have no limit. Without `ClientIpMiddleware` in the pipeline, the network of the address of the connection is used. To change what it does, extend the class and override the protected methods you need:

Method	Default	Use it to
<code>shouldProcess(\$request)</code>	<code>true</code>	Limit only some requests
<code>getIdentifier(\$request)</code>	Hash of the network of the client	Limit by something else, like an API key
<code>getTokensNeeded(\$request)</code>	<code>1</code>	Make some requests cost more
<code>getHeaders(\$limit)</code>	The two <code>X-RateLimit-*</code> headers	Change the headers sent

```
use Derafu\Http\Middleware\ThrottleMiddleware as BaseThrottleMiddleware;
use Psr\Http\Message\ServerRequestInterface;

class ThrottleMiddleware extends BaseThrottleMiddleware
{
    protected function shouldProcess(ServerRequestInterface $request): bool
    {
        // Only the API is limited.
        return str_starts_with($request->getUri()->getPath(), '/api');
    }
}
```

Custom Middlewares

Create your own by implementing PSR-15's `MiddlewareInterface` and adding it to the list:

```
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\MiddlewareInterface;
use Psr\Http\Server\RequestHandlerInterface;
```

```
class CustomMiddleware implements MiddlewareInterface
{
    public function process(
        ServerRequestInterface $request,
        RequestHandlerInterface $handler
    ): ResponseInterface {
        // Before the rest of the chain.
        $response = $handler->handle($request);
        // After the rest of the chain.

        return $response;
    }
}
```

Keep in mind where it goes:

- **Before RequestFactoryMiddleware** the request is a plain PSR-7 request, not a `Derafu\Http\Request`.
- **Before RouterMiddleware** `$request->route()` is not available yet.
- **After DispatcherMiddleware** the handler already ran, and the response is in the `derafu.response` attribute.

Last updated on 08/10/2026