Spring Cloud Gateway

Table of Contents

2.0.0.BUILD-SNAPSHOT

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

1. How to Include Spring Cloud Gateway

To include Spring Cloud Gateway in your project use the starter with group `org.springframework.cloud` and artifact id `spring-cloud-starter-gateway`. See the Spring Cloud Project page for details on setting up your build system with the current Spring

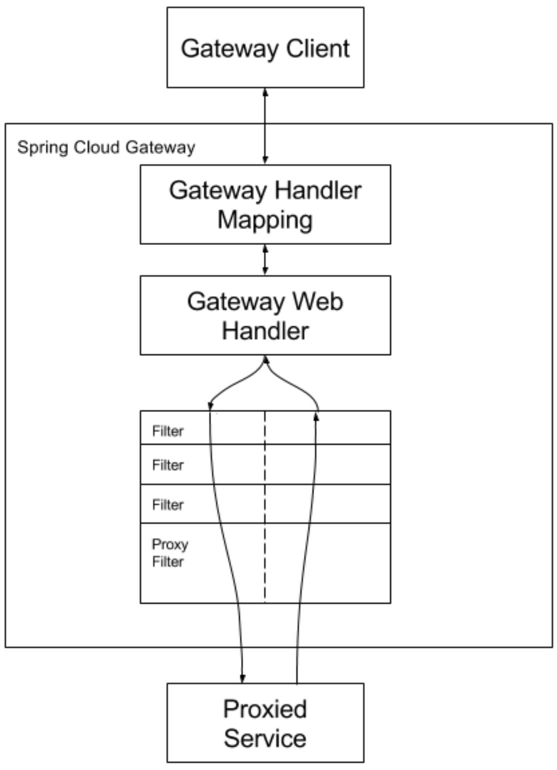
Cloud Release Train.

If you include the starter, but, for some reason, you do not want the gateway to be enabled, set `spring.cloud.gateway.enabled=false`.

2. Glossary

- **Route**: Route the basic building block of the gateway. It is defined by an ID, a destination URI, a collection of predicates and a collection of filters. A route is matched if aggregate predicate is true.
- **Predicate**: This is a Java 8 Function Predicate. The input type is a Spring Framework `ServerWebExchange`. This allows developers to match on anything from the HTTP request, such as headers or parameters.
- **Filter**: These are instances Spring Framework `GatewayFilter` constructed in with a specific factory. Here, requests and responses can be modified before or after sending the downstream request.

3. How It Works



Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a Route, it is sent to the Gateway Web Handler. This handler runs sends the request through a filter chain that is specific to the request. The reason the filters are divided by the dotted line, is that filters may execute logic before the proxy request is sent or after. All "pre" filter logic is executed, then the proxy request is made. After the proxy request is made, the "post" filter logic is executed.

4. Route Predicate Factories

Spring Cloud Gateway matches routes as part of the Spring WebFlux `HandlerMapping` infrastructure. Spring Cloud Gateway includes many built-in Route Predicate Factories. All of these predicates match on different attributes of the HTTP request. Multiple Route Predicate Factories can be combined and are combined via logical `and`.

4.1 After Route Predicate Factory

The After Route Predicate Factory takes one parameter, a datetime. This predicate matches requests that happen after the current datetime.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: after_route
        uri: http://example.org
        predicates:
        - After=2017-01-20T17:42:47.789-07:00[America/Denver]
```

This route matches any request after Jan 20, 2017 17:42 Mountain Time (Denver).

4.2 Before Route Predicate Factory

The Before Route Predicate Factory takes one parameter, a datetime. This predicate matches requests that happen before the current datetime.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: before_route
        uri: http://example.org
        predicates:
        - Before=2017-01-20T17:42:47.789-07:00[America/Denver]
```

This route matches any request before Jan 20, 2017 17:42 Mountain Time (Denver).

4.3 Between Route Predicate Factory

The Between Route Predicate Factory takes two parameters, datetime1 and datetime2. This predicate matches requests that happen after datetime1 and before datetime2. The datetime2 parameter must be after datetime1.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: between_route
        uri: http://example.org
        predicates:
        - Betweeen=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]
```

This route matches any request after Jan 20, 2017 17:42 Mountain Time (Denver) and before Jan 21, 2017 17:42 Mountain Time (Denver). This could be useful for maintenance windows.

4.4 Cookie Route Predicate Factory

The Cookie Route Predicate Factory takes two parameters, the cookie name and a regular expression. This predicate matches cookies that have the given name and the value matches the regular expression.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
```

```
      # =====================================
      - id: cookie_route
        uri: http://example.org
        predicates:
        - Cookie=chocolate, ch.p
```

This route matches the request has a cookie named `chocolate` who's value matches the `ch.p` regular expression.

4.5 Header Route Predicate Factory

The Header Route Predicate Factory takes two parameters, the header name and a regular expression. This predicate matches with a header that has the given name and the value matches the regular expression.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: header_route
        uri: http://example.org
        predicates:
        - Header=X-Request-Id, \d+
```

This route matches if the request has a header named `X-Request-Id` whos value matches the `\d+` regular expression (has a value of one or more digits).

4.6 Host Route Predicate Factory

The Host Route Predicate Factory takes one parameter: the host name pattern. The pattern is an Ant style pattern with `.` as the separator. This predicates matches the `Host` header that matches the pattern.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: host_route
        uri: http://example.org
        predicates:
        - Host=**.somehost.org
```

This route would match if the request has a `Host` header has the value `www.somehost.org` or `beta.somehost.org`.

4.7 Method Route Predicate Factory

The Method Route Predicate Factory takes one parameter: the HTTP method to match.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: method_route
        uri: http://example.org
        predicates:
        - Method=GET
```

This route would match if the request method was a `GET`.

4.8 Path Route Predicate Factory

The Path Route Predicate Factory takes one parameter: a Spring `PathMatcher` pattern.

application.yml.

```yaml
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: host_route
        uri: http://example.org
        predicates:
        - Path=/foo/{segment}
```

This route would match if the request path was, for example: `/foo/1` or `/foo/bar`.

This predicate extracts the URI template variables (like `segment` defined in the example above) as a map of names and values and places it in the `ServerWebExchange.getAttributes()` with a key defined in `PathRoutePredicate.URL_PREDICATE_VARS_ATTR`. Those values are then available for use by GatewayFilter Factories

4.9 Query Route Predicate Factory

The Query Route Predicate Factory takes two parameters: a required `param` and an optional `regexp`.

application.yml.

```yaml
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: query_route
        uri: http://example.org
        predicates:
        - Query=baz
```

This route would match if the request contained a `baz` query parameter.

application.yml.

```yaml
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: query_route
        uri: http://example.org
        predicates:
        - Query=foo, ba.
```

This route would match if the request contained a `foo` query parameter whose value matched the `ba.` regexp, so `bar` and `baz` would match.

4.10 RemoteAddr Route Predicate Factory

The RemoteAddr Route Predicate Factory takes a list (min size 1) of CIDR-notation strings, e.g. `192.168.0.1/16` (where `192.168.0.1` is an IP address and `16` is a subnet mask.

application.yml.

```yaml
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: remoteaddr_route
        uri: http://example.org
        predicates:
        - RemoteAddr=192.168.1.1/24
```

This route would match if the remote address of the request was, for example, `192.168.1.10`.

5. GatewayFilter Factories

Route filters allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. Route filters are scoped to a particular route. Spring Cloud Gateway includes many built-in GatewayFilter Factories.

5.1 AddRequestHeader GatewayFilter Factory

The AddRequestHeader GatewayFilter Factory takes a name and value parameter.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: add_request_header_route
        uri: http://example.org
        filters:
        - AddRequestHeader=X-Request-Foo, Bar
```

This will add `X-Request-Foo:Bar` header to the downstream request's headers for all matching requests.

5.2 AddRequestParameter GatewayFilter Factory

The AddRequestParameter GatewayFilter Factory takes a name and value parameter.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: add_request_parameter_route
        uri: http://example.org
        filters:
        - AddRequestParameter=foo, bar
```

This will add `foo=bar` to the downstream request's query string for all matching requests.

5.3 AddResponseHeader GatewayFilter Factory

The AddResponseHeader GatewayFilter Factory takes a name and value parameter.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: add_request_header_route
        uri: http://example.org
        filters:
        - AddResponseHeader=X-Response-Foo, Bar
```

This will add `X-Response-Foo:Bar` header to the downstream response's headers for all matching requests.

5.4 Hystrix GatewayFilter Factory

The Hystrix GatewayFilter Factory takes a single `name` parameters, which is the name of the `HystrixCommand`. (More options might be added in future releases).

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: hytstrix_route
        uri: http://example.org
        filters:
        - Hystrix=myCommandName
```

This wraps the remaining filters in a `HystrixCommand` with command name `myCommandName`.

5.5 PrefixPath GatewayFilter Factory

The PrefixPath GatewayFilter Factory takes a single `prefix` parameter.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: prefixpath_route
        uri: http://example.org
        filters:
        - PrefixPath=/mypath
```

This will prefix `/mypath` to the path of all matching requests. So a request to `/hello`, would be sent to `/mypath/hello`.

5.6 RequestRateLimiter GatewayFilter Factory

The RequestRateLimiter GatewayFilter Factory takes three parameters: `replenishRate`, `burstCapacity` & `keyResolverName`.

`replenishRate` is how many requests per second do you want a user to be allowed to do.

`burstCapacity` TODO: document burst capacity

`keyResolver` is a bean that implements the `KeyResolver` interface. In configuration, reference the bean by name using SpEL. `#{@myKeyResolver}` is a SpEL expression referencing a bean with the name `myKeyResolver`.

KeyResolver.java.

```
public interface KeyResolver {
	Mono<String> resolve(ServerWebExchange exchange);
}
```

The `KeyResolver` interface allows pluggable strategies to derive the key for limiting requests. In future milestones, there will be some `KeyResolver` implementations.

The redis implementation is based off of work done at Stripe. It requires the use of the `spring-boot-starter-data-redis-reactive` Spring Boot starter.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: requestratelimiter_route
        uri: http://example.org
        filters:
        - RequestRateLimiter=10, 20, #{@userKeyResolver}
```

Config.java.

```
@Bean
KeyResolver userKeyResolver() {
    return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));
}
```

This defines a request rate limit of 10 per user. The `KeyResolver` is a simple one that gets the `user` request parameter (note: this is not recommended for production).

5.7 RedirectTo GatewayFilter Factory

The RedirectTo GatewayFilter Factory takes a `status` and a `url` parameter. The status should be a 300 series redirect http code, such as 301. The url should be a valid url. This will be the value of the `Location` header.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: prefixpath_route
        uri: http://example.org
        filters:
        - RedirectTo=302, http://acme.org
```

This will send a status 302 with a `Location:http://acme.org` header to perform a redirect.

5.8 RemoveNonProxyHeaders GatewayFilter Factory

The RemoveNonProxyHeaders GatewayFilter Factory removes headers from forwarded requests. The default list of headers that is removed comes from the IETF.

The default removed headers are:

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailer
- Transfer-Encoding
- Upgrade

To change this, set the `spring.cloud.gateway.filter.remove-non-proxy-headers.headers` property to the list of header names to remove.

5.9 RemoveRequestHeader GatewayFilter Factory

The RemoveRequestHeader GatewayFilter Factory takes a `name` parameter. It is the name of the header to be removed.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: removerequestheader_route
        uri: http://example.org
        filters:
        - RemoveRequestHeader=X-Request-Foo
```

This will remove the `X-Request-Foo` header before it is sent downstream.

5.10 RemoveResponseHeader GatewayFilter Factory

The RemoveResponseHeader GatewayFilter Factory takes a `name` parameter. It is the name of the header to be removed.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: removeresponseheader_route
        uri: http://example.org
        filters:
        - RemoveResponseHeader=X-Response-Foo
```

This will remove the `X-Response-Foo` header from the response before it is returned to the gateway client.

5.11 RewritePath GatewayFilter Factory

The RewritePath GatewayFilter Factory takes a path `regexp` parameter and a `replacement` parameter. This uses Java regular expressions for a flexible way to rewrite the request path.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: rewritepath_route
        uri: http://example.org
        predicates:
        - Path=/foo/**
        filters:
        - RewritePath=/foo/(?<segment>.*), /$\{segment}
```

For a request path of `/foo/bar`, this will set the path to `/bar` before making the downstream request. Notice the `$\` which is replaced with `$` because of the YAML spec.

5.12 SecureHeaders GatewayFilter Factory

The SecureHeaders GatewayFilter Factory adds a number of headers to the response at the reccomendation from this blog post.

The following headers are added (allong with default values):

- `X-Xss-Protection:1; mode=block`
- `Strict-Transport-Security:max-age=631138519`
- `X-Frame-Options:DENY`
- `X-Content-Type-Options:nosniff`
- `Referrer-Policy:no-referrer`
- `Content-Security-Policy:default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; objec`
- `X-Download-Options:noopen`
- `X-Permitted-Cross-Domain-Policies:none`

To change the default values set the appropriate property in the `spring.cloud.gateway.filter.secure-headers` namespace:

Property to change:

- `xss-protection-header`
- `strict-transport-security`
- `frame-options`
- `content-type-options`
- `referrer-policy`
- `content-security-policy`
- `download-options`
- `permitted-cross-domain-policies`

5.13 SetPath GatewayFilter Factory

The SetPath GatewayFilter Factory takes a path `template` parameter. It offers a simple way to manipulate the request path by allowing templated segments of the path. This uses the uri templates from Spring Framework. Multiple matching segments are allowed.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: setpath_route
        uri: http://example.org
        predicates:
        - Path=/foo/{segment}
        filters:
        - SetPath=/{segment}
```

For a request path of `/foo/bar`, this will set the path to `/bar` before making the downstream request.

5.14 SetResponseHeader GatewayFilter Factory

The SetResponseHeader GatewayFilter Factory takes `name` and `value` parameters.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: setresponseheader_route
        uri: http://example.org
        filters:
        - SetResponseHeader=X-Response-Foo, Bar
```

This GatewayFilter replaces all headers with the given name, rather than adding. So if the downstream server responded with a `X-Response-Foo:1234`, this would be replaced with `X-Response-Foo:Bar`, which is what the gateway client would receive.

5.15 SetStatus GatewayFilter Factory

The SetStatus GatewayFilter Factory takes a single `status` parameter. It must be a valid Spring `HttpStatus`. It may be the integer value `404` or the string representation of the enumeration `NOT_FOUND`.

application.yml.

```
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: setstatusstring_route
        uri: http://example.org
        filters:
        - SetStatus=BAD_REQUEST
      - id: setstatusint_route
        uri: http://example.org
        filters:
        - SetStatus=401
```

In either case, the HTTP status of the response will be set to 401.

6. Global Filters

The `GlobalFilter` interface has the same signature as `GatewayFilter`. These are special filters that are conditionally applied to all routes. (This interface and usage are subject to change in future milestones).

6.1 Combined Global Filter and GatewayFilter Ordering

TODO: document ordering

6.2 Forward Routing Filter

The `ForwardRoutingFilter` looks for a URI in the exchange attribute `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the url has a `forward` scheme (ie `forward:///localendpoint`), it will use the Spring `DispatcherHandler` to handler the request. The unmodified original url is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute.

6.3 LoadBalancerClient Filter

The `LoadBalancerClientFilter` looks for a URI in the exchange attribute `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR`. If the url has a `lb` scheme (ie `lb://myservice`), it will use the Spring Cloud `LoadBalancerClient` to resolve the name (`myservice` in the previous example) to an actual host and port and replace the URI in the same attribute. The unmodified original url is appended to the list in the `ServerWebExchangeUtils.GATEWAY_ORIGINAL_REQUEST_URL_ATTR` attribute.

6.4 Netty Routing Filter

The Netty Routing Filter runs if the url located in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `http` or `https` scheme. It uses the Netty `HttpClient` to make the downstream proxy request. The response is put in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute for use in a later filter. (There is an experimental `WebClientHttpRoutingFilter` that performs the same function, but does not require netty)

6.5 Netty Write Response Filter

The `NettyWriteResponseFilter` runs if there is a Netty `HttpClientResponse` in the `ServerWebExchangeUtils.CLIENT_RESPONSE_ATTR` exchange attribute. It is run after all other filters have completed and writes the proxy response back to the gateway client response. (There is an experimental `WebClientWriteResponseFilter` that performs the same function, but does not require netty)

6.6 RouteToRequestUrl Filter

The `RouteToRequestUrlFilter` runs if there is a `Route` object in the `ServerWebExchangeUtils.GATEWAY_ROUTE_ATTR` exchange attribute. It creates a new URI, based off of the request URI, but updated with the URI attribute of the `Route` object. The new URI is placed in the ``` `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR ``` exchange attribute`.

6.7 Websocket Routing Filter

The Websocket Routing Filter runs if the url located in the `ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR` exchange attribute has a `ws` or `wss` scheme. It uses the Spring Web Socket infrastructure to forward the Websocket request downstream.

7. Configuration

Configuration for Spring Cloud Gateway is driven by a collection of `RouteDefinitionLocator`s.

RouteDefinitionLocator.java.

```
public interface RouteDefinitionLocator {
        Flux<RouteDefinition> getRouteDefinitions();
}
```

By default, a `PropertiesRouteDefinitionLocator` loads properties using Spring Boot's `@ConfigurationProperties` mechanism.

The configuration examples above all use a shortcut notation that uses positional arguments rather than named ones. The two examples below are equivalent:

application.yml.

```yaml
spring:
  cloud:
    gateway:
      routes:
      # =====================================
      - id: setstatus_route
        uri: http://example.org
        filters:
        - name: SetStatus
          args:
            status: 401
      - id: setstatusshortcut_route
        uri: http://example.org
        filters:
        - SetStatus=401
```

For some usages of the gateway, properties will be adequate, but some production use cases will benefit from loading configuration from an external source, such as a database. Future milestone versions will have `RouteDefinitionLocator` implementations based off of Spring Data Repositories such as: Redis, MongoDB and Cassandra.

7.1 Fluent Java Routes API

To allow for simple configuration in Java, there is a fluent API defined in the `Routes` class.

GatewaySampleApplication.java.

```java
// static imports from GatewayFilters and RoutePredicates
@Bean
public RouteLocator customRouteLocator(ThrottleGatewayFilterFactory throttle) {
    return Routes.locator()
            .route("test")
                .predicate(host("**.abc.org").and(path("/image/png")))
                .addResponseHeader("X-TestHeader", "foobar")
                .uri("http://httpbin.org:80")
            .route("test2")
                .predicate(path("/image/webp"))
                .add(addResponseHeader("X-AnotherHeader", "baz"))
                .uri("http://httpbin.org:80")
            .route("test3")
                .order(-1)
                .predicate(host("**.throttle.org").and(path("/get")))
                .add(throttle.apply(tuple().of("capacity", 1,
                     "refillTokens", 1,
                     "refillPeriod", 10,
                     "refillUnit", "SECONDS")))
                .uri("http://httpbin.org:80")
            .build();
}
```

This style also allows for more custom predicate assertions. The predicates defined by `RouteDefinitionLocator` beans are combined using logical `and`. By using the fluent Java API, you can use the `and()`, `or()` and `negate()` operators on the `Predicate` class.

8. Actuator API

TODO: document the `/gateway` actuator endpoint

9. Developer Guide

TODO: overview of writing custom integrations

9.1 Writing Custom Route Predicate Factories

TODO: document writing Custom Route Predicate Factories

9.2 Writing Custom GatewayFilter Factories

TODO: document writing Custom GatewayFilter Factories

9.3 Writing Custom Global Filters

TODO: document writing Custom Global Filters

9.4 Writing Custom Route Locators and Writers

TODO: document writing Custom Route Locators and Writers

10. Building a Simple Gateway Using Spring MVC

Spring Cloud Gateway provides a utility object called `ProxyExchange` which you can use inside a regular Spring MVC handler as a method parameter. It supports basic downstream HTTP exchanges via methods that mirror the HTTP verbs, or forwarding to a local handler via the `forward()` method.

Example (proxying a request to "/test" downstream to a remote server):

```
@RestController
@SpringBootApplication
public class GatewaySampleApplication {

        @Value("${remote.home}")
        private URI home;

        @GetMapping("/test")
        public ResponseEntity<?> proxy(ProxyExchange<Object> proxy) throws Exception {
                return proxy.uri(home.toString() + "/image/png").get();
        }

}
```

There are convenience methods on the `ProxyExchange` to enable the handler method to discover and enhance the URI path of the incoming request. For example you might want to extract the trailing elements of a path to pass them downstream:

```
@GetMapping("/proxy/path/**")
public ResponseEntity<?> proxyPath(ProxyExchange<?> proxy) throws Exception {
  String path = proxy.path("/proxy/path/");
  return proxy.uri(home.toString() + "/foos/" + path).get();
}
```

All the features of Spring MVC are available to Gateway handler methods. So you can inject request headers and query parameters, for instance, and you can constrain the incoming requests with declarations in the mapping annotation. See the documentation for `@RequestMapping` in Spring MVC for more details of those features.

Headers can be added to the downstream response using the `header()` methods on `ProxyExchange`.

You can also manipulate response headers (and anything else you like in the response) by adding a mapper to the `get()` etc. method. The mapper is a `Function` that takes the incoming `ResponseEntity` and converts it to an outgoing one.

First class support is provided for "sensitive" headers ("cookie" and "authorization" by default) which are not passed downstream, and for "proxy" headers (`x-forwarded-*`).