# 1.0 Envoy Introduction

In the Envoy introduction module, you’ll get an overview of Envoy and walk through Envoy building blocks using an example. By the end of this module, you’ll understand Envoy by running it with an example configuration.

# 1.1 What is Envoy?

The industry is moving toward microservice architectures and cloud-native solutions. With hundreds and thousands of microservices developed using different technologies, these systems can become complex and hard to debug.

As an application developer, you’re thinking about business logic -- purchasing a product or generating an invoice. However, any business logic like that will result in multiple service calls between different services. Each of the services probably has its timeouts, retry logic, and other network-specific code that might have to be tweaked or fine-tuned.

If at any point the initial request fails, it will be hard to trace it through multiple services, pinpoint where the failure occurred, and understand why it failed. Was the network unreliable? Do we need to adjust retries or timeouts? Or is it a business logic issue or a bug?

Adding to this complexity of debugging is the fact that services might use inconsistent tracing and logging mechanisms. These issues make it hard to identify the problem, where it happened, and how to fix it. This is especially true if you’re an application developer and debugging network issues falls outside of your core skills.

What would make debugging these network issues easier is to push networking concerns out of the applications stack and have another component deal with the networking part. This is what Envoy can do.

In one of its deployment patterns, we have an Envoy instance running next to every service instance. This type of deployment is also called a **sidecar deployment**. The other pattern Envoy works well with is the **edge-proxy**, which is used to build API gateways.

The Envoy and the application form an atomic entity but are still separate processes. The application deals with the business logic, and Envoy deals with network concerns.

In case of a failure, separating concerns makes it easier to determine if the failure is coming from the application or the network.

To help with network debugging , Envoy provides the following high-level features:

## Out-of-process architecture

Envoy is a self-contained process designed to run alongside every application -- the sidecar deployment pattern we mentioned earlier. The collection of centrally configured Envoys forms a transparent mesh.

The responsibility of routing and other network features gets pushed to Envoy. Applications send requests to a virtual address (localhost) instead of a real one (like a public IP address or a hostname), unaware of the network topology. Applications no longer bear the responsibility of routing as that task is delegated to an external process.

Instead of having applications manage their network configuration, the network configuration is managed independently of the application at the Envoy level.In an organization, this can free app developers to focus on the business logic of the application.

Envoy works with any programming language. You can write your applications in Go, Java, C++, or any other language, and Envoy can bridge the gap between them. Its behavior is identical, regardless of the application’s programming language or the operating system they are running on.

Envoy can also be deployed and upgraded transparently across the entire infrastructure. This is compared to deploying library upgrades for each separate app, which can be extremely painful and time-consuming.

The out-of-process architecture is beneficial as it gives us consistency across programming languages/applications stacks, and we get an independent lifecycle and all of the Envoy networking features for free, without having to individually address them in every application.

## L3/L4 filter architecture

Envoy is an L3/L4 network proxy that makes decisions based on IP addresses and TCP or UDP ports. It features a plugable filter chain to write your filters to perform different TCP/UDP tasks.

A **filter chain** comes from a shell idea where the output of one operation is piped into another operation. For example:

ls -l | grep "Envoy\*.cc" | wc -l

Envoy can construct logic and behavior by stacking desired filters that form a filter chain. Many filters exist and support tasks such as raw TCP proxy, UDP proxy, HTTP proxy, TLS client cert authentication, etc. Envoy is also extensible, and we can write our filters.

## L7 filter architecture

Envoy supports an additional HTTP L7 filter layer. We can insert HTTP filters into the HTTP connection management subsystem that performs different tasks such as buffering, rate limiting, routing/forwarding, etc.

## First-class HTTP/2 support

Envoy supports both HTTP/1.1 and HTTP/2 and can operate as a transparent HTTP/1.1 to HTTP/2 proxy in both directions. This means that any combination of HTTP/1.1 and HTTP/2 clients and target servers can be bridged. Even if your legacy applications aren't communicating via HTTP/2, if you deploy them alongside Envoy proxy, they'll end up communicating via HTTP/2.

The recommended service-to-service configuration uses HTTP/2 between all Envoys to create a mesh of persistent connections that requests and responses can be multiplexed over.

## HTTP routing

When operating in HTTP mode and using REST, Envoy supports a routing subsystem capable of routing and redirecting requests based on path, authority, content type, and runtime values. This functionality is useful when using Envoy as a front/edge proxy for building API gateways and leveraged when building a service mesh (sidecar deployment pattern).

## gRPC ready

Envoy supports all HTTP/2 features required to be used as the routing and load balancing substrate for gRPC requests and responses.

gRPC is an open-source remote procedure call (RPC) system that uses HTTP/2 for transport and protocol buffers as the interface description language (IDL), and that provides features such as authentication, bidirectional streaming, and flow control, blocking/nonblocking bindings, and cancellation and timeouts.

## Service discovery and dynamic configuration

We can configure Envoy using static configuration files that describe the services and how to communicate with them.

For advanced scenarios where statically configuring Envoy would be impractical, Envoy supports dynamic configuration and automatically reloads configuration at runtime. A set of discovery services called xDS can be used to dynamically configure Envoy through the network and provide Envoy information about hosts, clusters HTTP routing, listening sockets, and cryptographic material. At that time, Envoy will attempt to gracefully drain all connections.

## Health checking

One feature associated with load balancers is routing traffic only to healthy and available upstream services. Envoy supports a health checking subsystem that performs active health checks of upstream service clusters. Envoy then uses the union of service discovery and health checking information to determine healthy load balancing targets. Envoy can also support passive health checking via an outlier detection subsystem.

## Advanced load balancing

Envoy supports automatic retries, circuit breaking, global rate limiting (using an external rate-limiting service), request shadowing (or traffic mirroring), outlier detection, and request hedging.

## Front/edge proxy support

Envoy has features that make it well suited to run as an edge proxy. Such features include TLS termination, HTTP/1.1, HTTP/2, and HTTP/3 support, and HTTP L7 routing.

## TLS termination

The decoupling of the application and the proxy enables TLS termination (mutual TLS) between all services in the mesh deployment model.

## Best-in-class observability

For observability, Envoy generates logs, metrics, and traces. Envoy currently supports [statsd](https://github.com/etsy/statsd) (and compatible providers) as the statistics sink for all subsystems. Thanks to extensibility, we can also plug in different statistics providers if needed.

## HTTP/3 (Alpha)

Envoy 1.19.0 supports HTTP/3 upstream and downstream and translates between HTTP/1.1, HTTP/2, and HTTP/3 in either direction.

# 1.2 Envoy building blocks

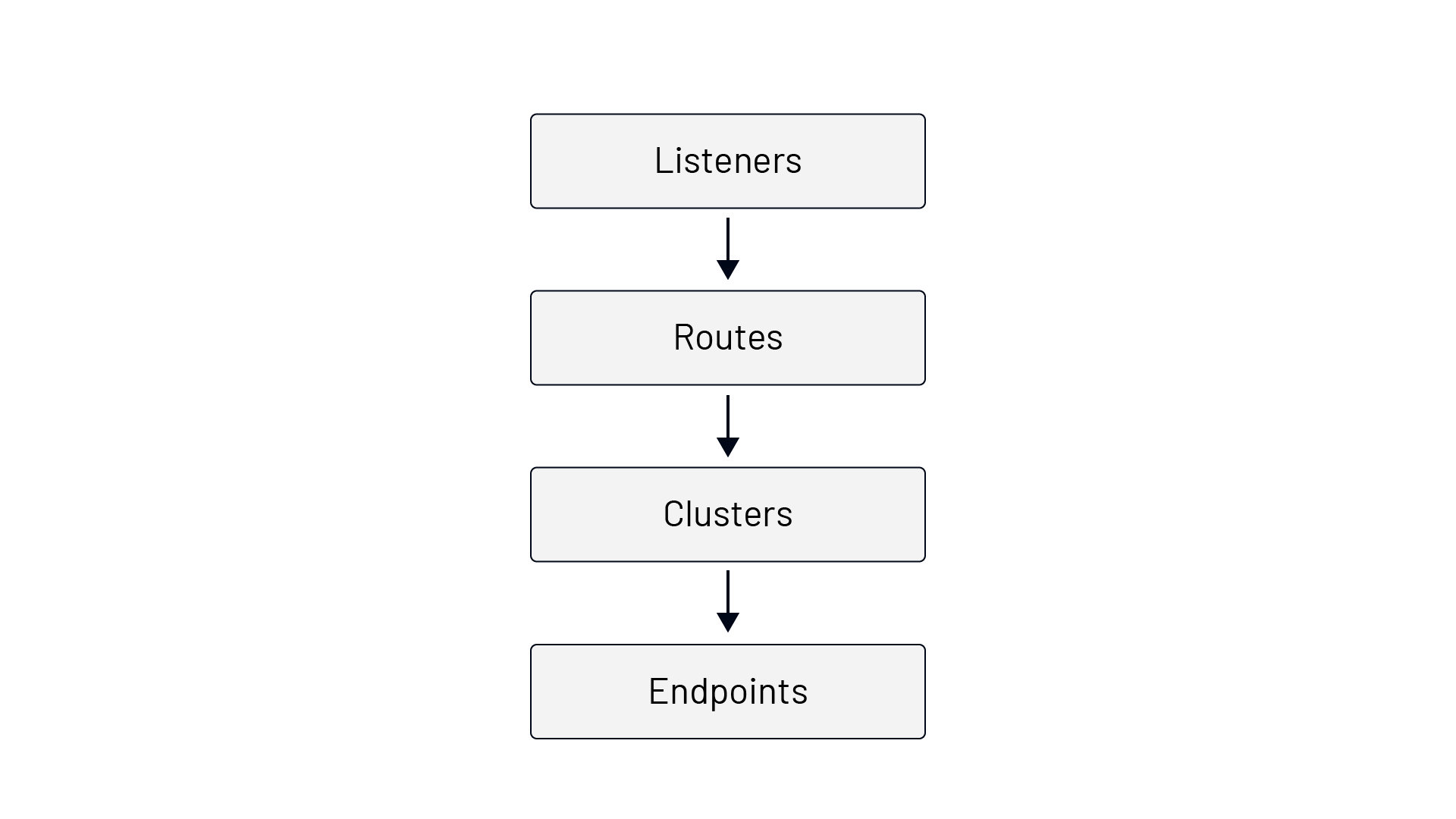
In this lesson, we'll explain the basic building blocks of Envoy.

The root of the Envoy configuration is called bootstrap configuration. It contains fields where we can provide the static or dynamic resources and high-level Envoy configuration (e.g., Envoy instance names, runtime configuration, enable the administrative interface, and so on).

To get started, we'll mainly focus on the static resources, and later in the course, we'll introduce how to configure dynamic resources.

Envoy outputs numerous statistics, depending on enabled components and their configuration. We'll mention different stats throughout the course, and we'll talk more about statistics in a dedicated module later in the course.

The diagram below shows the request flow through these concepts:



It all starts with the **listeners**. Envoy exposes listeners that are named network locations, either an IP address and a port or a Unix Domain Socket path. Envoy receives connections and requests through listeners. Consider the following Envoy configuration:

static\_resources:  
 listeners:  
 - name: listener\_0  
 address:  
 socket\_address:  
 address: 0.0.0.0  
 port\_value: 10000  
 filter\_chains: [{}]

With the Envoy config above, we're declaring a listener called listener\_0 on address 0.0.0.0 and port 10000. That means Envoy is listening on 0.0.0.0:10000 for incoming requests.

There are different pieces to configure for each listener. However, the only required setting is the address. The above configuration is valid, and you can run Envoy with it -- albeit it's not useful as all connections will be closed.

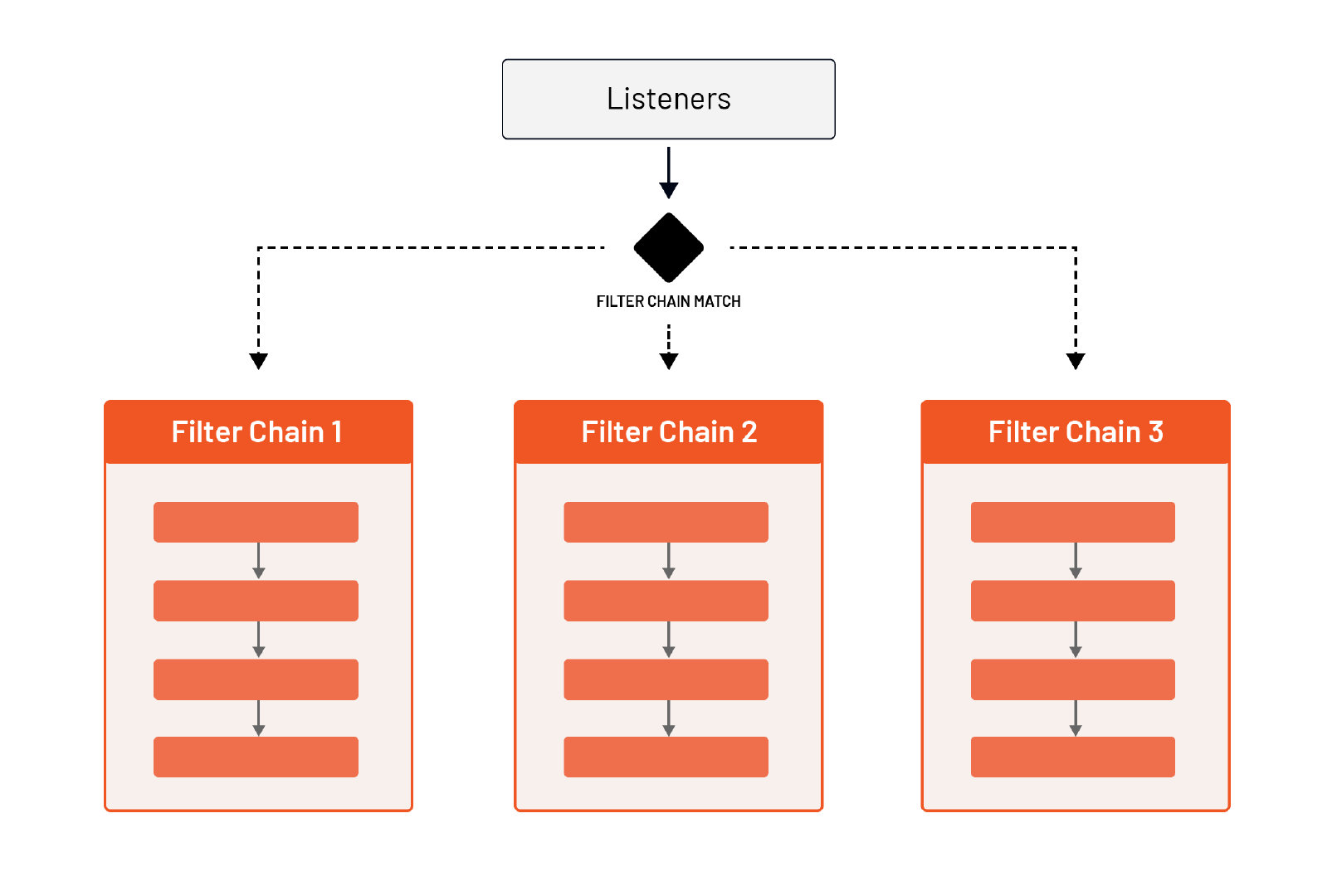
We're leaving the filter\_chains field empty because no additional operations are required after receiving the packet.

To move to the next building block (routes), we need to create one or more network filter chains (filter\_chains) with at least one filter.

The network filters usually operate on the packet's payload by looking at the payload and parsing it. For example, a Postgres network filter parses the packet's body and checks the kind of database operation or the result it carries.

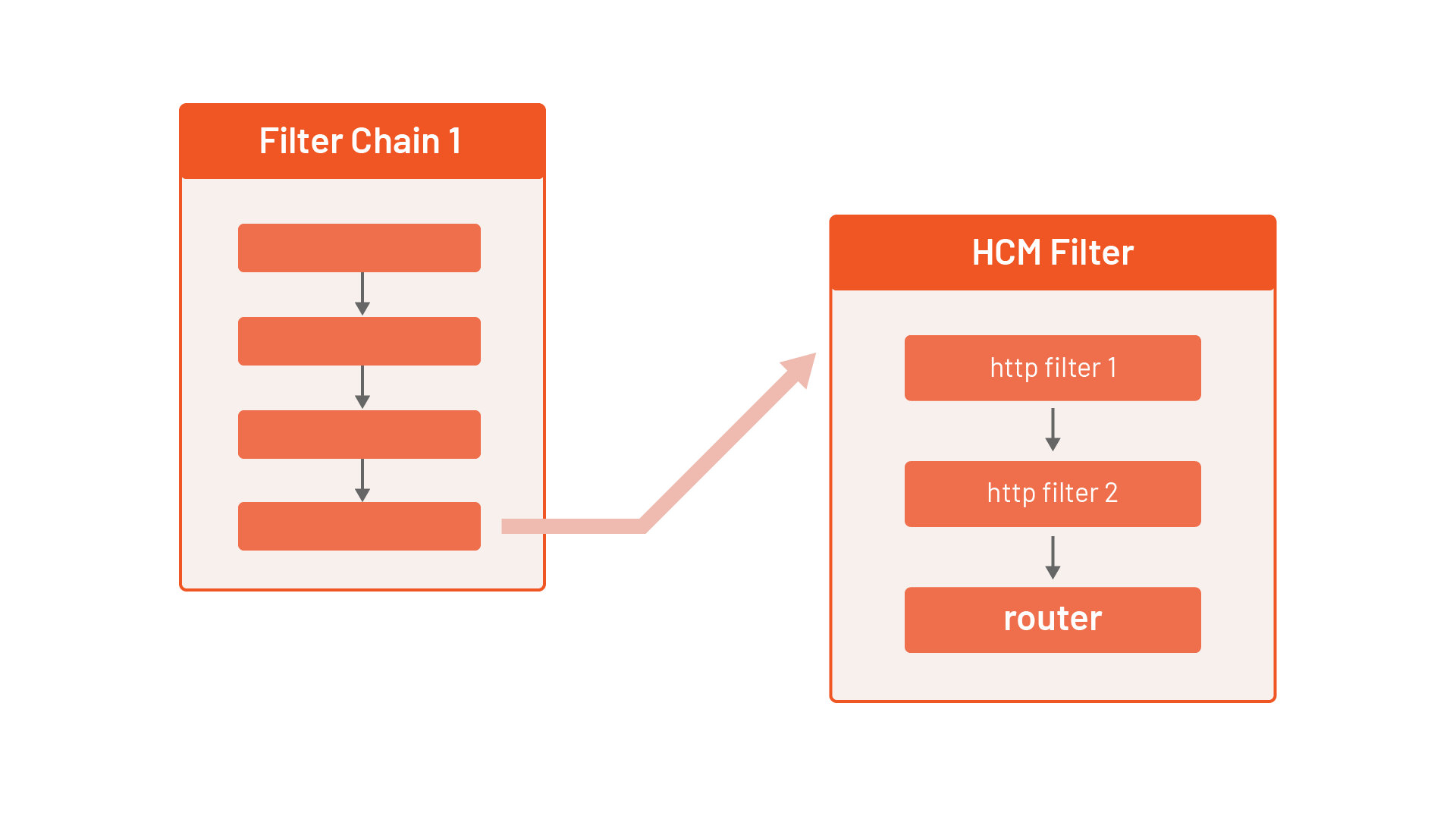
Envoy defines three categories of filters: listener filters, network filters, and HTTP filters. The listener filters kick in right after a packet has been received and usually operate on the packet's headers. The examples of listener filters are the proxy listener filter (which extracts the PROXY protocol header), or the TLS inspector listener filter (checks if the traffic is TLS and, if it is, extracts data from the TLS handshake).

Each request that comes in through a listener can flow through multiple filters. We can also write a configuration that selects a different filter chain based on the incoming request or connection properties.



One special, built-in network filter is called **HTTP connection manager** filter or **HCM**. The HCM filter is capable of translating from raw bytes to HTTP-level messages. It can handle access logging, generate request IDs, manipulate headers, manage route tables, and collect statistics. We'll go into more details about HCM in later lessons.

Like we can have multiple network filters defined (one of them being the HCM) for each listener, Envoy also supports defining multiple HTTP-level filters within the HCM filter. We can define these HTTP filters under the field called http\_filters.



The last filter in the HTTP filter chain must be the router filter (envoy.filters.HTTP.router). The router filter is responsible for performing routing tasks. This finally brings us to the second building block -- the **routes**.

We define the route configuration in the HCM filter under the route\_config field. Within the route configuration, we can match the incoming requests by looking at the metadata (URI, headers, ...) and, based on that, define where traffic is sent.

A top-level element in the routing configuration is a virtual host. Each virtual host has a name that's used when emitting statistics (not used for routing) and a set of domains that get routed to it.

Let's consider the following route configuration and the set of domains:

route\_config:  
 name: my\_route\_config  
 virtual\_hosts:  
 - name: tetrate\_host  
 domains: ["tetrate.io"]  
 routes:  
 ...  
 - name: test\_hosts  
 domains: ["test.tetrate.io", "qa.tetrate.io"]  
 routes:  
 ...

If an incoming request's destination is tetrate.io (i.e., the Host/Authority header is set to one of the values), the routes defined in the tetrate\_hosts virtual hosts will get processed.

Similarly, if the Host/Authority header contains test.tetrate.io or qa.tetrate.io, the routes under the test\_hosts virtual host will be processed. Using this design, we could have a single listener (0.0.0.0:10000) to handle multiple top-level domains.

If you specify multiple domains in the array, the search order is the following:

1. Exact domain names (e.g. tetrate.io)
2. Suffix domain wildcards (e.g. \*.tetrate.io)
3. Prefix domain wildcards (e.g. tetrate.\*)
4. Special wildcard matching any domain (\*)

After Envoy matches the domain, it's time to process the routes field within the selected virtual host. This is where we specify how to match a request and what to do next with the request (e.g., redirect, forward, rewrite, send a direct response etc.).

Let's look at an example:

static\_resources:  
 listeners:  
 - name: listener\_0  
 address:  
 socket\_address:  
 address: 0.0.0.0  
 port\_value: 10000  
 filter\_chains:  
 - filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: hello\_world\_service  
 http\_filters:  
 - name: envoy.filters.http.router  
 route\_config:  
 name: my\_first\_route  
 virtual\_hosts:  
 - name: direct\_response\_service  
 domains: ["\*"]  
 routes:  
 - match:  
 prefix: "/"  
 direct\_response:  
 status: 200  
 body:  
 inline\_string: "yay"

The top portion of the configuration is the same as we saw before. We've added the HCM filter, the statistics prefix (hello\_world\_service), a single HTTP filter (router), and the route configuration.

Within the virtual hosts, we're matching any domain. Under routes, we match the prefix (/), and then we can send a response.

We have multiple options when it comes to matching a request:

|  |  |  |
| --- | --- | --- |
| Route match | Description | Example |
| prefix | The prefix must match the beginning of the :path header | /hello matches hello.com/hello, hello.com/helloworld, and hello.com/hello/v1 |
| path | The path must exactly match the :path header | /hello matches hello.com/hello, but not hello.com/helloworld or hello.com/hello/v1 |
| safe\_regex | The provided regex must match the :path header | /\d{3} matches any 3 digit number after /. For example, hello.com/123, but not hello.com/hello or hello.com/54321 |
| connect\_matcher | Matcher only matches CONNECT requests |  |

Once Envoy matches the request to a route, we can route it, redirect it, or return a direct response. We're using the **direct response** through the direct\_response configuration field in this example.

You can save the above configuration to envoy-direct-response.yaml.

We'll use a CLI called [func-e](https://func-e.io/) to run Envoy. func-e allows us to select and use different Envoy versions.

We can download func-e CLI by running the following command:

curl https://func-e.io/install.sh | sudo bash -s -- -b /usr/local/bin

Now we run Envoy with the configuration we created:

func-e run -c envoy-direct-response.yaml

Once Envoy starts, we can send a request to localhost:10000 to get back the direct response we configured:

$ curl localhost:10000  
yay

Similarly, if we added a different host header (for example -H "Host: hello.com") would get the same response back because the hello.com host matches the domains defined in the virtual host.

In most cases, sending a direct response from configuration is a nice feature, but we'd have a set of endpoints or hosts we route the traffic to. The way to do that in Envoy is by defining **clusters**.

Clusters are a group of similar upstream hosts that accept the traffic. This could be a list of hosts or IP addresses on which your services are listening.

For example, let's say our hello world service is listening on 127.0.0.0:8000. Then, we can create a cluster with a single endpoint like this:

clusters:  
- name: hello\_world\_service  
 load\_assignment:  
 cluster\_name: hello\_world\_service  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 8000

Clusters are defined at the same level as listeners using the clusters field. We use the cluster when referencing a cluster in the route configuration and when emitting statistics. The name has to be unique across all clusters.

Under the load\_assignment field, we can define the list of endpoints to load balance to, together with the load balancing policy settings.

Envoy supports multiple load-balancing algorithms (round-robin, Maglev, least-request, random) configured from a combination of static bootstrap configuration, DNS, dynamic xDS (CDS and EDS services), and active/passive health checks. If we don't explicitly set a load balancing algorithm through the lb\_policy field, it defaults to round-robin.

The endpoints field defines a group of endpoints that belong to a specific locality. Using the optional locality field, we could specify where the upstream hosts are running and then use that during load balancing (i.e., proxy requests to an endpoint closer to the caller).

Adding new endpoints instructs the load balancer to distribute the traffic among more than one recipient. Usually, the load balancer treats all endpoints equally, but the cluster definition allows for building a hierarchy within endpoints.

For example, endpoints may have a **weight** attribute, which will instruct the load balancer to send more/less traffic to such endpoints compared to other endpoints.

The other hierarchy type is based on **locality** and is usually used to define failover architecture. This hierarchy allows us to define "preferred" endpoints that are geographically closer, as well as "backup" endpoints that should be used in the case when "preferred" endpoints become unhealthy.

Since we only have a single endpoint, we haven't set a locality. The place to define an actual endpoint that Envoy can route traffic to is under the lb\_endpoints field.

Clusters configuration is also the place where we can configure the following optional features:

* active health checking (health\_checks)
* circuit breakers (circuit\_breakers)
* outlier detection (outlier\_detection)
* additional protocol options when handling HTTP requests upstream
* set of optional network filters to be applied to all outgoing connections and more.

Like the listener's address, the endpoint address can either be a socket address or a Unix Domain Socket. In our case, we're using a socket address and defining the endpoint for our service at 127.0.0.1:8000. Once the endpoint is selected, the request is proxied upstream to the endpoint.

Let's see how the cluster we defined comes together with the rest of the configuration:

static\_resources:  
 listeners:  
 - name: listener\_0  
 address:  
 socket\_address:  
 address: 0.0.0.0  
 port\_value: 10000  
 filter\_chains:  
 - filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: hello\_world\_service  
 http\_filters:  
 - name: envoy.filters.http.router  
 route\_config:  
 name: my\_first\_route  
 virtual\_hosts:  
 - name: direct\_response\_service  
 domains: ["\*"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 cluster: hello\_world\_service  
 clusters:  
 - name: hello\_world\_service  
 connect\_timeout: 5s  
 load\_assignment:  
 cluster\_name: hello\_world\_service  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 8000

We've added the cluster configuration, and instead of using the direct\_response, we are using the route field and specifying the cluster name.

To try out this configuration, let's start a hello-world Docker image on port 8000:

docker run -dit -p 8000:3000 gcr.io/tetratelabs/hello-world:1.0.0

We can send a request to 127.0.0.1:8000 to check that we get back a "Hello World" response.

Next, let's save the above Envoy configuration to envoy-clusters.yaml and start the Envoy proxy:

func-e run -c envoy-cluster.yaml

When the Envoy proxy starts, send a request to 0.0.0.0:10000 to have Envoy proxy the request to the hello world endpoint:

$ curl -v 0.0.0.0:10000  
...  
> GET / HTTP/1.1  
> Host: localhost:10000  
> User-Agent: curl/7.64.0  
> Accept: \*/\*  
>  
< HTTP/1.1 200 OK  
< date: Wed, 30 Jun 2021 23:53:47 GMT  
< content-length: 11  
< content-type: text/plain; charset=utf-8  
< x-envoy-upstream-service-time: 0  
< server: envoy  
<  
\* Connection #0 to host localhost left intact  
Hello World

# 1.3 Lab prerequisites

From the verbose output, we'll notice the response headers x-envoy-upstream-service-time and server: envoy that are set by the Envoy proxy.

During the course we'll use practical labs to practice what we've learned.

All labs were developed and tested on Linux Debian 10 with the following tools installed:

* [Docker](https://docker.com)
* [Python 3](https://www.python.org/downloads/)
* [func-e](https://func-e.io/)
* [hey](https://github.com/rakyll/hey)

# 2.0 HTTP connection manager (HCM)

In the HCM module, we'll expand on the HTTP connection manager filter. We'll learn about filter ordering and how HTTP routing and matching work. We'll show you how to split traffic, manipulate headers, configure timeouts, and implement retries, request mirroring, and rate-limiting.

By the end of this module, you'll have a good understanding of the HCM filter and how to route and split HTTP traffic, manipulate headers, and more.

# 2.1 HTTP connection manager (HCM) introduction

HCM is a network-level filter that translates raw bytes into HTTP-level messages and events (e.g., headers received, body data received, etc.).

The HCM filter also handles standard HTTP functionality. It supports features such as access logging, request ID generation and tracing, header manipulation, route table management, and statistics.

From the protocol perspective, HCM natively supports HTTP/1.1, WebSockets, HTTP/2, and HTTP/3 (still in Alpha).

Envoy proxy was designed to be an HTTP/2 multiplexing proxy, as reflected in the terminology used for describing Envoy's components.

**HTTP/2 terminology**

In HTTP/2, a *stream* is a bidirectional flow of bytes within an established connection. Each stream can carry one or more *messages*. A message is a complete sequence of *frames* that map to an HTTP request or response message. Finally, a *frame* is the smallest unit of communication in HTTP/2. Each frame contains a frame header which at a minimum identifies the stream to which the frame belongs. A frame can carry information about HTTP headers, message payload, and so on.

Regardless of which connection a stream originates from (HTTP/1.1, HTTP/2, or HTTP/3), Envoy uses a feature called *codec API* to translate from different wire protocols into a protocol-agnostic model of streams, requests, responses, and so on. The protocol-agnostic model means that most Envoy code doesn't need to understand the specifics of each protocol.

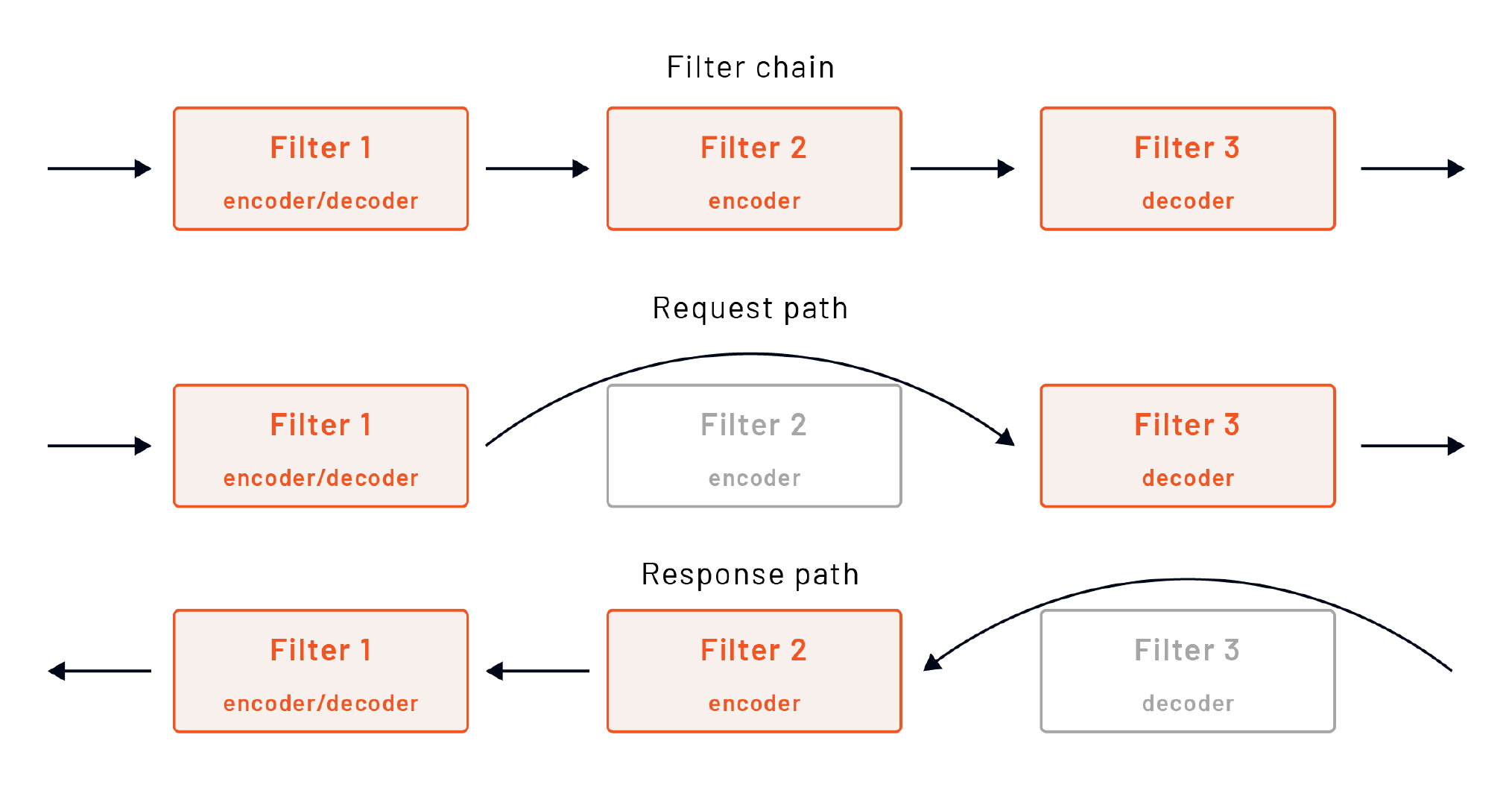
## HTTP filters

Within the HCM, Envoy supports a set of HTTP filters. Unlike the listener level filters, these filters operate on HTTP-level messages without knowing the underlying protocol (HTTP/1.1, HTTP/2, etc.) or multiplexing capabilities.

There are three types of HTTP filters:

* Decoder: invoked when HCM is decoding parts of the request stream
* Encoder: invoked when HCM is encoding parts of the response stream
* Decoder/encoder: invoked on both paths, decoding, and encoding

The figure below explains how Envoy invokes different filter types on request and response paths.



Like the network filters, individual HTTP filters can either stop or continue executing subsequent filters and share states among themselves within the context of a single request stream.

### Data sharing

At a high level, we can break the data sharing between filters into **static** and **dynamic** states.

The static state contains any immutable data set when Envoy loads the configuration, and it's broken into three sections:

**1. Metadata**

Envoy configuration such as listeners, routes, or clusters contains a metadata field that stores key/value pairs. Metadata allows us to store filter-specific configurations. The values can't change (they are immutable) and get shared across all requests/connections. For example, metadata values get used when using subset selectors in clusters.

**2. Typed metadata**

Instead of converting metadata to a typed class object for each stream or request, the typed metadata allows filters to register a one-time conversion logic for a specific key. The metadata coming from xDS gets converted to class objects at configuration load time, and filters can request the typed version at runtime, without conversion each time.

**3. HTTP per-route filter configuration** We can also specify configuration per virtual host or route compared to the global configuration that applies to all virtual hosts. Per-route configuration is embedded into the route table and can be specified under the typed\_per\_filter\_config field.

The other way to share data is using **dynamic state**. Dynamic state gets generated per connection or HTTP stream, and it is mutable by the filter that produces it. An object called StreamInfo provides a way to store and retrieve typed objects from a map.

### Filter ordering

The order in which HTTP filters are specified matters. Consider the following HTTP filter chain:

http\_filters:  
 - filter\_1  
 - filter\_2  
 - filter\_3

Typically, the last filter in the chain will usually be the router filter. Assuming all filters are decoder/encoder filters, the order in which HCM invokes them on the request path is filter\_1, filter\_2, filter\_3.

On the response path, Envoy invokes only encoder filters, but in the reverse order. Since all three filters are decoder/encoder filters, the order on the response path is filter\_3, filter\_2, filter\_1.

### Built-in HTTP filters

Envoy already comes with several built-in HTTP filters, such as CORS, CSRF, health check, JWT authentication, and others. You can find the complete list of HTTP filters [here](https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters#config-http-filters).

# 2.2 HTTP routing

The previously mentioned router filter (envoy.filters.http.router) is the one that implements HTTP forwarding. The router filter is used in almost all HTTP proxy scenarios. The main job of the router filter is to look at the route table and to route (forward and redirect) the requests accordingly.

The router uses the information from the incoming request (e.g., host or authority headers) and matches it to an upstream cluster through virtual hosts and routing rules.

All configured HTTP filters use the route configuration (route\_config) that contains the route table. Even though the primary consumer of the route table will be the router filter, other filters have access to it if they want to make any decisions based on the destination of the request.

A set of **virtual hosts** makes up the route configuration. Each virtual host has a logical name, a set of domains that can get routed to it based on the request header, and a set of routes that specify how to match a request and indicate what to do next.

Envoy also supports priority routing at the route level. Each priority has its connection pool and circuit-breaking settings. The two currently supported priorities are DEFAULT and HIGH. The priority defaults to DEFAULT if we don't explicitly provide it.

Here's a snippet that shows an example of a route configuration:

route\_config:  
 name: my\_route\_config # Name used for stats, not relevant for routing  
 virtual\_hosts:  
 - name: bar\_vhost  
 domains: ["bar.io"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 priority: HIGH  
 cluster: bar\_io  
 - name: foo\_vhost  
 domains: ["foo.io"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 cluster: foo\_io  
 - match:  
 prefix: "/api"  
 route:  
 cluster: foo\_io\_api

When an HTTP request comes in, the virtual host, domain, and route matching happen in order:

1. The host or authority header gets matched to a value specified in each virtual host's domains field. For example, virtual host foo\_vhost matches if the host header is set to foo.io.
2. The entries under routes within the matched virtual host are checked next. If a match is found, no further checks are made, and a cluster is selected. For example, if we matched the foo.io virtual host and the request prefix is /api, the cluster foo\_io\_api is selected.
3. If provided, each virtual cluster (virtual\_clusters) in a virtual host is checked for a match. If there's a match, a virtual cluster is used, and no further virtual cluster checks are made.

Virtual cluster is a way of specifying regex matching rules against specific endpoints and explicitly generating stats for the matched request.

The order of virtual hosts as well as routes within each host matters. Consider the following route configuration:

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/api"  
 route:  
 cluster: hello\_io\_api  
 - match:  
 prefix: "/api/v1"  
 route:  
 cluster: hello\_io\_api\_v1

Which route/cluster is selected if we send the following request?

curl hello.io/api/v1

The first route that sets the cluster hello\_io\_api is matched. That's because matches are evaluated in order by the prefix. However, we might have wrongly expected the route with the prefix /api/v1 to be matched. To workaround this issue, we could swap the order of routes or use a different matching rule.

# 2.3 Request Matching

## Path matching

We've only talked about one matching rule that matches a prefix using the prefix field. The table below explains the other supported matching rules.

|  |  |
| --- | --- |
| Rule name | Description |
| prefix | Prefix must match the beginning of the path header. For example, prefix /api will match paths /api and /api/v1, but not /. |
| path | Path must match the exact path header (without query string). For example, path /api will match paths /api, but not /api/v1 or /. |
| safe\_regex | Path must match the specified regular expression. For example, regex ^/products/\d+$ will match path /products/123 or /products/321 but not /products/hello or /api/products/123 |
| connect\_matcher | Matcher only matches CONNECT requests (currently in Alpha) |

By default, the prefix and path matching is case-sensitive. To make it case insensitive, we can set the case\_sensitive to false. Note that this setting doesn't apply for safe\_regex matching.

## Headers matching

Another way of matching requests is by specifying a set of headers. The router checks the request headers against all specified headers in the route config. The match is made if all specified headers exist on the request and have the same values set.

Multiple matching rules can be applied to headers.

**Range match**

The range\_match checks if the request header value is within the specified integer range in the base ten notation. The value can include an optional plus or minus sign, followed by digits.

To use the range matching, we specify the start and the end of the range. The start value is inclusive, while the end of the range is exclusive ([start, end)).

- match:  
 prefix: "/"  
 headers:  
 - name: minor\_version  
 range\_match:  
 start: 1  
 end: 11

The above range match will match the minor\_version header values if it's set to any number between 1 and 10.

**Present match**

The present\_match checks for a presence of a specific header in the incoming request.

- match:  
 prefix: "/"  
 headers:  
 - name: debug  
 present\_match: true

The above snippet will evaluate to true if we set the debug header, regardless of the header value. If we set the present\_match value to false, we can check for the absence of the header.

**String match**

The string\_match allows us to match the exact header values, by prefix or suffix, using regular expression or checking if the value contains a specific string.

- match:  
 prefix: "/"  
 headers:  
 # Header `regex\_match` matches the provided regular expression  
 - name: regex\_match  
 string\_match:  
 safe\_regex\_match:  
 google\_re2: {}  
 regex: "^v\\d+$"  
 # Header `exact\_match` contains the value `hello`  
 - name: exact\_match  
 string\_match:  
 exact: "hello"  
 # Header `prefix\_match` starts with `api`  
 - name: prefix\_match  
 string\_match:  
 prefix: "api"  
 # Header `suffix\_match` ends with `\_1`  
 - name: suffix\_match  
 string\_match:  
 suffix: "\_1"  
 # Header `contains\_match` contains the value "debug"  
 - name: contains\_match  
 string\_match:  
 contains: "debug"

**Invert match**

If we set the invert\_match, the match result is inverted.

- match:  
 prefix: "/"  
 headers:  
 - name: version  
 range\_match:   
 start: 1  
 end: 6  
 invert\_match: true

The above snippet will check that the value in the version header falls between 1 and 5; however, because we added the invert\_match field, it inverts the result and checks if the header values falls out of that range.

The invert\_match can be used by other matchers. For example:

- match:  
 prefix: "/"  
 headers:  
 - name: env  
 contains\_match: "test"  
 invert\_match: true

The above snippet will check that the env header value doesn't contain the string test. If we set the env header and it doesn't include the string test, the whole match evaluates to true.

## Query parameters matching

Using the query\_parameters field, we can specify the parameters from the URL query on which the route should match. The filter will check the query string from the path header and compare it against the provided parameters.

If there's more than one query parameter specified, they must match the rule to evaluate to true.

Consider the following example:

- match:  
 prefix: "/"  
 query\_parameters:  
 - name: env  
 present\_match: true

The above snippet will evaluate to true if there's a query parameter called env set. It's not saying anything about the value. It's just checking for its presence. For example, the following request would evaluate to true using the above matcher:

GET /hello?env=test

We can also use a string matcher to check for the values of query parameters. The table below lists different rules for string matching.

|  |  |
| --- | --- |
| Rule name | Description |
| exact | Must match the exact value of the query parameter. |
| prefix | Prefix must match the beginning of the query parameter value. |
| suffix | Suffix must match the ending of the query parameter value. |
| safe\_regex | Query parameter value must match the specified regular expression. |
| contains | Checks if the query parameter value contains a specific string. |

In addition to the above rules, we can also use the ignore\_case field to indicate whether the exact, prefix, or suffix matching should be case sensitive or not. If set to true, the matching is case-insensitive.

Here's another example of a case-insensitive query parameter matching using the prefix rule:

- match:  
 prefix: "/"  
 query\_parameters:  
 - name: env  
 string\_match:  
 prefix: "env\_"  
 ignore\_case: true

The above will evaluate to true if there's a query parameter called env whose value starts with env\_. For example, env\_staging and ENV\_prod evaluates to true.

## gRPC and TLS matchers

We can configure the other two matchers on the routes: the gRPC route matcher (grpc) and TLS context matcher (tls\_context).

The gRPC matcher will only match on the gRPC requests. The router checks the content-type header for application/grpc and other application/grpc+ values to determine if the request is a gRPC request.

For example:

- match:  
 prefix: "/"  
 grpc: {}

Note the gRPC matcher doesn't have any options.

The above snippet will match the route if the request is a gRPC request.

Similarly, the TLS matcher, if specified, will match the TLS context against provided options. Within the tls\_context field, we can define two boolean values -- presented and validated. The presented field checks whether a certificate is presented or not. The validated field checks whether a certificate is validated or not.

For example:

- match:  
 prefix: "/"  
 tls\_context:  
 presented: true  
 validated: true

The above match evaluates to true if a certificate is both presented and validated.

# 2.4 Traffic splitting

Envoy supports traffic splitting to different routes within the same virtual host. We can split the traffic between two or more upstream clusters.

There are two different approaches. The first one uses the percentages specified in the runtime object, and the second one uses weighted clusters.

## Traffic splitting using runtime percentages

Using the percentages from the runtime object lends itself well for the canary release or progressive delivery scenarios. In this scenario, we want to shift traffic from one upstream cluster to another gradually.

The way to achieve this is by providing a runtime\_fraction configuration. Let's use an example to explain how traffic splitting using runtime percentages works.

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/"  
 runtime\_fraction:  
 default\_value:  
 numerator: 90  
 denominator: HUNDRED  
 route:  
 cluster: hello\_v1  
 - match:  
 prefix: "/"  
 route:  
 cluster: hello\_v2

The above configuration declares two versions of the hello service: hello\_v1 and hello\_v2.

In the first match, we're configuring the runtime\_fraction field by specifying a numerator (90) and a denominator (HUNDRED). Envoy calculates the final fractional value using the numerator and the denominator. In this case the final value is 90% (90/100 = 0.9 = 90%).

Envoy generates a random number within the range [0, denominator) (e.g. [0, 100] in our case). If the random number is less than the numerator value, the router matches the route and sends the traffic to cluster hello\_v1 in our case.

If the random number is greater than the numerator, Envoy continues to evaluate the remaining match criteria. Since we have the exact prefix match for the second route, it matches, and Envoy sends the traffic to cluster hello\_v2. Once we set the numerator value to 0, none of the random numbers will be greater than the numerator value. Hence all traffic will go to the second route.

We can also set the denominator value in a runtime key. For example:

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/"  
 runtime\_fraction:  
 default\_value:  
 numerator: 0  
 denominator: HUNDRED  
 runtime\_key: routing.hello\_io  
 route:  
 cluster: hello\_v1  
 - match:  
 prefix: "/"  
 route:  
 cluster: hello\_v2  
...  
layered\_runtime:  
 layers:  
 - name: static\_layer  
 static\_layer:  
 routing.hello\_io: 90

In this example, we're specifying a runtime key called routing.hello\_io. We can set the value for that key under the layered runtime field in the configuration -- this could also be read and updated dynamically either from a file or through a runtime discovery service (RTDS). For simplicity's sake, we're setting it directly in the config file.

When Envoy does the matching this time, it will see that the runtime\_key is provided and will use that value instead of the numerator value. With the runtime key, we don't have to hard-code the value in the configuration, and we can have Envoy read it from a separate file or RTDS.

The approach with runtime percentages works well when you have two clusters. Still, it becomes complicated when you want to split traffic to more than two clusters or if you're running A/B testing or multivariate testing scenarios.

## Traffic splitting using weighted clusters

The weighted clusters approach is ideal when you're splitting traffic between two or more versions of the service. In this approach, we assign different weights for multiple upstream clusters. Whereas the method with runtime percentages uses numerous routes, we only need a single route for the weighted clusters.

We'll talk more about upstream clusters in the next module. To explain the traffic splitting with weighted clusters, we can think of an upstream cluster as a collection of endpoints traffic can be sent to.

We specify multiple weighted clusters (weighted\_clusters) within the route instead of setting a single cluster (cluster).

Continuing with the previous example, this is how we could re-write the configuration to use weighted clusters instead:

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 weighted\_clusters:  
 clusters:  
 - name: hello\_v1  
 weight: 90  
 - name: hello\_v2  
 weight: 10

Under the weighted clusters, we could also set the runtime\_key\_prefix that will read the weights from the runtime key configuration. Note that Envoy uses the weights next to each cluster if the runtime key configuration is not there.

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 weighted\_clusters:  
 runtime\_key\_prefix: routing.hello\_io  
 clusters:  
 - name: hello\_v1  
 weight: 90  
 - name: hello\_v2  
 weight: 10  
...  
layered\_runtime:  
 layers:  
 - name: static\_layer  
 static\_layer:  
 routing.hello\_io.hello\_v1: 90  
 routing.hello\_io.hello\_v2: 10

The weight represents the percentage of the traffic Envoy sends to the upstream cluster. The sum of all weights has to be 100. However, using the total\_weight field, we can control the value the sum of all weights has to equal to. For example, the following snippet sets the total\_weight to 15:

route\_config:  
 virtual\_hosts:  
 - name: hello\_vhost  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 weighted\_clusters:  
 runtime\_key\_prefix: routing.hello\_io  
 total\_weight: 15  
 clusters:  
 - name: hello\_v1  
 weight: 5  
 - name: hello\_v2  
 weight: 5  
 - name: hello\_v3  
 weight: 5

To dynamically control the weights, we can set the runtime\_key\_prefix. The router uses the runtime key prefix value to construct the runtime keys associated with each cluster. If we provide the runtime key prefix, the router will check the runtime\_key\_prefix + "." + cluster\_name value, where cluster\_name denotes the entry in the clusters array (e.g. hello\_v1, hello\_v2). If Envoy doesn't find the runtime key, it will use the value specified in the configuration as the default value.

# 2.5 Header manipulation

HCM supports manipulating request and response headers at the weighted cluster, route, virtual host, and/or global configuration level.

Note that we can't modify all headers directly from the configuration. The exception is if we use a Wasm extension. Then, we could modify the :authority header for example.

The immutable headers are the pseudo-headers (prefixed by :, such as :scheme) and the host header. Additionally, headers such as :path and :authority can be indirectly modified through prefix\_rewrite, regex\_rewrite, and host\_rewrite configuration.

Envoy applies the headers to requests/responses in the following order:

1. Weighted cluster-level headers
2. Route-level headers
3. Virtual host-level headers
4. Global-level headers

The order means that Envoy might overwrite a header set on the weighted cluster level by headers configured at the higher level (route, virtual host, or global).

At each level, we can set the following fields to add/remove request/response headers:

* response\_headers\_to\_add: array of headers to add to the response
* response\_headers\_to\_remove: array of headers to remove from the response
* request\_headers\_to\_add: array of headers to add to the request
* request\_headers\_to\_remove: array of headers to remove from the request

In addition to hardcoding the header values, we can also use variables to add dynamic values to the headers. The variable names get delimited by the percent symbol (%). The list of supported variable names includes %DOWNSTREAM\_REMOTE\_ADDRESS%, %UPSTREAM\_REMOTE\_ADDRESS%, %START\_TIME%, %RESPONSE\_FLAGS% and many more. You can find the completed list of variables [here](https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_conn_man/headers#custom-request-response-headers).

Let's look at an example that shows how to add/remove headers from request/response at different levels:

route\_config:  
 response\_headers\_to\_add:  
 - header:   
 key: "header\_1"  
 value: "some\_value"  
 # If true (default) it appends the value to existing values,  
 # otherwise it replaces the existing value  
 append: false  
 response\_headers\_to\_remove: "header\_we\_dont\_need"  
 virtual\_hosts:  
 - name: hello\_vhost  
 request\_headers\_to\_add:  
 - header:   
 key: "v\_host\_header"  
 value: "from\_v\_host"  
 domains: ["hello.io"]  
 routes:  
 - match:  
 prefix: "/"  
 route:  
 cluster: hello  
 response\_headers\_to\_add:  
 - header:   
 key: "route\_header"  
 value: "%DOWNSTREAM\_REMOTE\_ADDRESS%"  
 - match:  
 prefix: "/api"  
 route:  
 cluster: hello\_api  
 response\_headers\_to\_add:  
 - header:   
 key: "api\_route\_header"  
 value: "api-value"  
 - header:  
 key: "header\_1"  
 value: "this\_will\_be\_overwritten"

## Standard headers

Envoy manipulates a set of headers when the request is received (decoding) and when it sends the request to the upstream cluster (encoding).

When using bare-bone Envoy configuration to route the traffic to a single cluster, the following headers are set during the encoding:

':authority', 'localhost:10000'  
':path', '/'  
':method', 'GET'  
':scheme', 'http'  
'user-agent', 'curl/7.64.0'  
'accept', '\*/\*'  
'x-forwarded-proto', 'http'  
'x-request-id', '14f0ac76-128d-4954-ad76-823c3544197e'  
'x-envoy-expected-rq-timeout-ms', '15000'

On encoding (response), a different set of headers is sent:

':status', '200'  
'x-powered-by', 'Express'  
'content-type', 'text/html; charset=utf-8'  
'content-length', '563'  
'etag', 'W/"233-b+4UpNDbOtHFiEpLMsDEDK7iTeI"'  
'date', 'Fri, 16 Jul 2021 21:59:52 GMT'  
'x-envoy-upstream-service-time', '2'  
'server', 'envoy'

The table below explains the different headers set by Envoy either during decoding or encoding.

|  |  |
| --- | --- |
| Header | Description |
| :scheme | Set and available to filters, and forwarded upstream. (For HTTP/1, :scheme header is set either from the absolute URL or from the x-forwaded-proto header value) |
| user-agent | Usually set by the client, but can be modified when add\_user\_agent is enabled (only if the header is not already set). The value gets determined by the --service-cluster command-line option. |
| x-forwarded-proto | Standard header for identifying the protocol that a client used to connect to the proxy. The value is either http or https. |
| x-request-id | Used by Envoy to uniquely identify a request, also used in access logging and tracing. |
| x-envoy-expected-rq-timeout-ms | Specifies the time in milliseconds the router expects the request to be completed. This is read from x-envoy-upstream-rq-timeout-ms header value (assuming respect\_expected\_rq\_timeout is set) or from the route timeout setting (default is 15 seconds). |
| x-envoy-upstream-service-time | Time in milliseconds spent by the endpoint processing the request and the network latency between Envoy and upstream host. |
| server | Set to value specified in server\_name field (defaults to envoy). |

A slew of other headers get set or consumed by Envoy, depending on the scenarios. We'll call out different headers as we discuss these scenarios and features in the rest of the course.

## Header sanitization

Header sanitization is a process that involves adding, removing, or modifying request headers for security reasons. There are some headers Envoy will potentially sanitize:

|  |  |
| --- | --- |
| Header | Description |
| x-envoy-decorator-operation | Overrides any locally defined span name generated by the tracing mechanism. |
| x-envoy-downstream-service-cluster | Contains the service cluster of the caller (removed for external requests). Determined by the --service-cluster command line option requires user\_agent to be set to true. |
| x-envoy-downstream-service-node | Like the previous header, value is determined by the --service-node option. |
| x-envoy-expected-rq-timeout-ms | Specifies the time in milliseconds the router expects the request to be completed. This is read from x-envoy-upstream-rq-timeout-ms header value (assuming respect\_expected\_rq\_timeout is set) or from the route timeout setting (default is 15 seconds). |
| x-envoy-external-address | Trusted client address (see XFF below for details on how this is determined). |
| x-envoy-force-trace | Forces traces to be collected. |
| x-envoy-internal | Set to true if the request is internal (see XFF below for details on how this is determined). |
| x-envoy-ip-tags | Set by the HTTP IP tagging filter if the external address is defined in the IP tags. |
| x-envoy-max-retries | Maximum number of retries if retry policy is configured. |
| x-envoy-retry-grpc-on | Retries failed requests for specific gRPC status codes. |
| x-envoy-retry-on | Specifies the retry policy. |
| x-envoy-upstream-alt-stat-name | Emist upstream response code/timing stats to a dual stat tree. |
| x-envoy-upstream-rq-per-try-timeout-ms | Sets a per try timeout on routed requests. |
| x-envoy-upstream-rq-timeout-alt-response | If present, sets a 204 response code (instead of 504) in case of a request timeout. |
| x-envoy-upstream-rq-timeout-ms | Overrides the route configuration timeout. |
| x-forwarded-client-cert | Indicates certificate information of part of all of the clients/proxies that a request has flowed through. |
| x-forwarded-for | Indicates the IP addresses request went through. See XFF below for more details. |
| x-forwarded-proto | Sets the originating protocol (http or https). |
| x-request-id | Used by Envoy to uniquely identify a request. Also used in access logging and tracing. |

Whether to sanitize a specific header or not depends on where the request is coming from. Envoy determines whether the request is external or internal by looking at the x-forwarded-for header (XFF) and the internal\_address\_config setting.

## XFF

XFF or x-forwaded-for header indicates the IP addresses request went through on its way from the client to the server. Proxies between downstream and upstream services append the IP address of the nearest client to the XFF list before proxying the request.

Envoy doesn't automatically append the IP address to XFF. Envoy only appends the address if the use\_remote\_address (default is false) is set to true, and skip\_xff\_append is set to false.

When use\_remote\_address is set to true, the HCM uses the real remote address of the client connection when determining whether the origin is internal or external and when modifying headers. This value controls how Envoy determines the **trusted client address**.

**Trusted client address**

A trusted client address is the first source IP address that's known to be accurate. The source IP address of the downstream node that made a request to the Envoy proxy is considered correct.

Note that complete XFF sometimes cannot be trusted as malicious agents can forge it. However, if a trusted proxy puts the last address in the XFF, it can be trusted. For example, if we look at the request path IP1 -> IP2 -> IP3 -> Envoy, the IP3 is the node Envoy will consider accurate.

Envoy supports extensions set through the original\_ip\_detection\_extensions field to help with determining the original IP address. Currently, there are two extensions, custom\_header and xff.

With the custom header extension, we can provide a header name that contains the original downstream remote address. Additionally, we can also tell HCM to treat the detected address as trusted.

With the xff extension, we can specify the number of additional proxy hops starting from the right side of the x-forwarded-for header to trust. If we'd set this value to 1 and use the same example as above, the trusted addresses would be IP2 and IP3.

Envoy uses the trusted client address to determine if the request is internal or external. If we set the use\_remote\_address to true the request is considered internal if it doesn't contain XFF and the immediate downstream node's connection to Envoy has an internal source address. Envoy uses [RFC1918](https://datatracker.ietf.org/doc/html/rfc1918) or [RFC4193](https://datatracker.ietf.org/doc/html/rfc4193) to determine the internal source address.

If we set the use\_remote\_address to false (default value), the request is internal only if XFF contains a single internal source address as defined by the above two RFCs.

Let's look at a quick example and set the use\_remote\_address to true and skip\_xff\_append to false:

...  
- filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 use\_remote\_address: true  
 skip\_xff\_append: false  
 ...

If we send a request to the proxy from the same machine (i.e. internal request), the headers sent to the upstream will look like this:

':authority', 'localhost:10000'  
':path', '/'  
':method', 'GET'  
':scheme', 'http'  
'user-agent', 'curl/7.64.0'  
'accept', '\*/\*'  
'x-forwarded-for', '10.128.0.17'  
'x-forwarded-proto', 'http'  
'x-envoy-internal', 'true'  
'x-request-id', '74513723-9bbd-4959-965a-861e2162555b'  
'x-envoy-expected-rq-timeout-ms', '15000'

Most of these headers are the same as we saw in the standard headers example. However, two headers got added -- the x-forwarded-for and the x-envoy-internal. The x-forwarded-for will contain the internal IP address, and the x-envoy-internal header will get set because we used XFF to determine the address. Instead of figuring out if the request is internal or not by parsing the x-forwarded-for header, we check for the presence of the x-envoy-internal header to quickly determine whether the request is internal or external.

If we send a request from outside of that network, the following headers get sent to the upstream:

':authority', '35.224.50.133:10000'  
':path', '/'  
':method', 'GET'  
':scheme', 'http'  
'user-agent', 'curl/7.64.1'  
'accept', '\*/\*'  
'x-forwarded-for', '50.35.69.235'  
'x-forwarded-proto', 'http'  
'x-envoy-external-address', '50.35.69.235'  
'x-request-id', 'dc93fd48-1233-4220-9146-eac52435cdf2'  
'x-envoy-expected-rq-timeout-ms', '15000'

Notice the :authority value is an actual IP address, instead of just localhost. Similarly, the x-forwarded-for header contains the IP address of the called. There's no x-envoy-internal header because the request is external. However, we do get a new header called x-envoy-external-address. Envoy sets this header only for external requests. The header can be forwarded between internal services and used for the analytics based on the origin client's IP address.

# 2.6 Reply modification

The HCM supports modifying and customizing a response that gets returned by Envoy. Note that this doesn't work for responses returned by upstream.

Local replies are responses generated by Envoy. Local reply works by defining a set of **mappers** that allow filtering and changing responses. For example, Envoy sends a local HTTP 404 if there aren't any routes or upstream clusters defined.

Each mapper must have a filter defined that compares the request properties with specified values (for example, comparing if status code equals 403). We can choose from [multiple filters](https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/accesslog/v3/accesslog.proto#envoy-v3-api-msg-config-accesslog-v3-accesslogfilter) to match on status code, duration, headers, response flags, and more.

In addition to the filter field, the mapper has fields for a new status code (status\_code), body (body and body\_format\_override), and headers (headers\_to\_add). For example, we can have a filter that matches the request status code 403 and then change the status code to 500, update the body, or add headers.

Here's an example that rewrites an HTTP 503 response to HTTP 401. Note that this refers to the status code returned by Envoy. Envoy will return a 503 if the upstream doesn't exist, for example.

...  
- name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 local\_reply\_config:  
 mappers:  
 - filter:  
 status\_code\_filter:  
 comparison:  
 op: EQ  
 value:  
 default\_value: 503  
 runtime\_key: some\_key  
 headers\_to\_add:  
 - header:  
 key: "service"  
 value: "unavailable"  
 append: false  
 status\_code: 401  
 body:  
 inline\_string: "Not allowed"

Note the runtime\_key field is required. If Envoy can't find the runtime key, it falls back to the default\_value.

# 2.7 Request ID generation

Unique request IDs are crucial for tracing requests through multiple services, visualizing request flows, and pinpointing sources of latency.

We can configure how the request ID gets generated through the request\_id\_extension field. If we don't provide any configuration, Envoy uses the default extension called UuidRequestIdConfig.

The default extension generates a unique identifier (UUID4) and populates the x-request-id HTTP header. Envoy uses the 14th nibble of the UUID to determine what happens with the trace.

If the 14th nibble is set to 9, the tracing should be sampled. If it's set to a it should be forced traced due to server-side override (a), or if set to b it should force traced due to client-side request ID joining.

The reason the 14th nibble is chosen is that it's fixed to 4 by design. Therefore, 4 indicates a default UUID and no trace status, for example 7b674932-635d-\*\*4\*\*ceb-b907-12674f8c7267.

The two configuration options we have in the UuidRequestIdconfig are the pack\_trace\_reason and use\_request\_id\_for\_trace\_sampling.

...  
..  
 route\_config:  
 name: local\_route  
 request\_id\_extension:  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.request\_id.uuid.v3.UuidRequestIdConfig  
 pack\_trace\_reason: false  
 use\_request\_id\_for\_trace\_sampling: false  
 http\_filters:  
 - name: envoy.filters.http.router  
...

The pack trace reason is a boolean value that controls whether the implementation alters the UUID to contain the trace sampling decision as mentioned above. The default value is true. The use\_request\_id\_for\_trace\_sampling sets whether to use x-request-id for sampling or not. The default value is true as well.

# 2.8 Timeouts

Envoy supports numerous configurable timeouts that depend on the scenarios you're using the proxy for.

We'll look at the different configurable timeouts in the HCM section. Note that other filters and components have their separate timeouts, and we'll not cover them here.

Some of the timeouts that are set at a higher-levels in configuration -- for example, at the HCM level -- can be overwritten at the lower levels, such as the HTTP route level.

Probably the most well-known timeout is the request timeout. The request timeout (request\_timeout) specifies the amount of time Envoy waits for the entire request to be received (e.g., 120s). The timer is activated when the request gets initiated. The timer is deactivated when the last request byte gets sent upstream or when the response gets initiated. By default, the timeout is disabled if not provided or set to 0.

A similar timeout called idle\_timeout represents when a downstream or upstream connection gets terminated if there are no active streams. The default idle timeout is set to 1 hour. The idle timeout can be set in the common\_http\_protocol\_options in the HCM configuration as shown below:

...  
filters:  
- name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 common\_http\_protocol\_options:  
 # Set idle timeout to 10 minutes  
 idle\_timeout: 600s  
...

To configure the idle timeout for upstream connections, we can use the same field common\_http\_protocol\_options, but in the clusters section.

There's also a timeout that pertains to the headers called request\_headers\_timeout. This timeout specifies the amount of time Envoy wails for the request headers to be received (e.g. 5s). The timer gets activated upon receiving the first byte of the headers. The time is deactivated when the last byte of the headers is received. By default, the timeout is disabled if not provided or if set to 0.

Some other timeouts are available to set, such as the stream\_idle\_timeout, drain\_timeout, and delayed\_close\_timeout.

If we move down the hierarchy, the next stop is the route timeouts. As mentioned earlier, timeouts at the route level can overwrite the HCM timeouts and a couple of additional timeouts.

The route timeout is the time Envoy waits for the upstream to respond with a complete response. The timer starts once the entire downstream request has been received. The timeout defaults to 15 seconds; however, it isn't compatible with responses that never end (i.e., streaming). In that case, the timeout needs to be disabled, and stream\_idle\_timeout should be used instead.

We can use the idle\_timeout field to overwrite the stream\_idle\_timeout on the HCM level.

We can also mention the per\_try\_timeout setting. This timeout is used in connection with retries, and it specifies a timeout for each try. Usually, the individual tries should be using a shorter timeout than the value set by the timeout field.

# 2.9 Retries

We can define retry policies on the virtual host and the route level. A retry policy set on the virtual host level will apply for all routes in that virtual host. If there's a retry policy defined on the route level, it will take precedence over the virtual host policy, and it gets treated separately -- i.e. the route-level retry policy doesn't inherit values from the virtual host level retry policy. Even though Envoy treats the retry policies independently, the configuration is the same.

In addition to setting the retry policy in the configuration, we can also configure it through request headers (i.e., x-envoy-retry-on header).

Within the Envoy configuration, we can configure the following:

1. Maximum number of retries

Envoy will retry requests up to a configured maximum. The exponential backoff algorithm is the default used to determine the intervals between retries. Another way to determine the interval between retries is through the headers (e.g. x-envoy-upstream-rq-per-try-timeout-ms). All retries are also contained within the overall request timeout, the request\_timeout configuration setting. By default, Envoy sets the number of retries to one.

1. Retry conditions

We can retry the requests based on different conditions. For example, we can only retry 5xx response codes, gateway failures, 4xx response codes, and so on.

1. Retry budgets

Retry budget specifies a limit on concurrent requests in relation to the number of active requests. This can help prevent retry traffic contributing to the traffic volume.

1. Host selection retry plugins

The host selection during retries usually follows the same process as the original request did. Using a retry plugin, we can change this behavior and specify a host or priority predicate that will reject a specific host and cause the host selection to be reattempted.

Let's look at a couple of configuration examples on how to define a retry policy. We're using httpbin and matching the /status/500 path that returns the 500 response code.

route\_config:  
 name: 5xx\_route  
 virtual\_hosts:  
 - name: httpbin  
 domains: ["\*"]  
 routes:  
 - match:  
 path: /status/500  
 route:  
 cluster: httpbin  
 retry\_policy:  
 retry\_on: "5xx"  
 num\_retries: 5

Within the retry\_policy field, we're setting the retry condition (retry\_on) to 500, which means that we want to retry only if the upstream returns an HTTP 500 (which it will). Envoy will retry the request five times. This is configured via the num\_retries field.

If we run Envoy and send a request, the request will fail (HTTP 500), and the following log entry will be created:

[2021-07-26T18:43:29.515Z] "GET /status/500 HTTP/1.1" 500 URX 0 0 269 269 "-" "curl/7.64.0" "1ae9ffe2-21f2-43f7-ab80-79be4a95d6d4" "localhost:10000" "127.0.0.1:5000"

Notice the 500 URX section telling us that the upstream responded with 500, and the URX response flag means that Envoy rejected the request because the upstream retry limit was reached.

The retry condition can be set to one or more values, separated by a comma, specified in the table below.

|  |  |
| --- | --- |
| Retry condition (retry\_on) | Description |
| 5xx | Retry on 5xx response code or if the upstream doesn't respond (includes connect-failure and refused-stream) |
| gatewayerror | Retry on 502, 503, or 504 response codes. |
| reset | Retry if upstream doesn't respond at all. |
| connect-failure | Retry if request failes due to a connection failure to the upstream server (e.g. connect timeout). |
| envoy-ratelimited | Retry if x-envoy-ratelimited header is present. |
| retriable-4xx | Retry if upstream responds with a retriable 4xx response code (only HTTP 409 at the moment). |
| refused-stream | Retry if upstream resets the stream with a REFUSED\_STREAM error code. |
| retriable-status-codes | Retry if upstream responds with any response code matching one defined in the x-envoy-retriable-status-codes header (e.g. comma delimited list of integers, for example "502,409"). |
| retriable-headers | Retry if upstream response includes any headers matching in the x-envoy-retriable-header-names header. |

In addition to controlling the responses on which Envoy retries the request, we can also configure the host selection logic for the retries. We can specify the retry\_host\_predicate that Envoy uses when selecting a host for retries.

We can keep track of previously attempted hosts (envoy.retry\_host\_predicates.previous\_host) and reject them if they've already been attempted. Or, we can reject any hosts marked as canary hosts (e.g., any hosts marked with canary: true) using envoy.retry\_host\_predicates.canary\_hosts predicate.

For example, here's how to configure the previous\_hosts plugin to reject any previously attempted hosts and retry the host selection a maximum of 5 times:

route\_config:  
 name: 5xx\_route  
 virtual\_hosts:  
 - name: httpbin  
 domains: ["\*"]  
 routes:  
 - match:  
 path: /status/500  
 route:  
 cluster: httpbin  
 retry\_policy:  
 retry\_host\_predicate:  
 - name: envoy.retry\_host\_predicates.previous\_hosts  
 host\_selection\_retry\_max\_attempts: 5

With multiple endpoints in the cluster defined, we'd see the retries sent to different hosts with each retry.

## Request hedging

The idea behind request hedging is to send multiple requests simultaneously to different hosts and use the results from the upstream that responds first. Note that we usually configure this for idempotent requests where making the same call multiple times has the same effect.

We can configure the request hedging by specifying a hedge policy. Currently, Envoy performs hedging only in response to a request timeout. So when an initial request times out, a retry request is issued without canceling the original timed-out request. Envoy will return the first good response based on the retry policy to the downstream.

The hedging can be configured can be enabled by setting the hedge\_on\_per\_try\_timeout field to true. Just like the retry policy, it can be enabled on the virtual host or route level:

route\_config:  
 name: 5xx\_route  
 virtual\_hosts:  
 - name: httpbin  
 domains: ["\*"]  
 hedge\_policy:  
 hedge\_on\_per\_try\_timeout: true  
 routes:  
 - match:  
 ...

# 2.10 Request mirroring

Using a request mirroring policy (request\_mirroring\_policies) on the route level, we can configure Envoy to shadow traffic from one cluster to another.

Traffic shadowing or request mirroring is when incoming requests destined for one cluster are duplicated and sent to a second cluster. The mirrored requests are "fire and forget", meaning that Envoy doesn't wait for the shadow cluster to respond before sending the response from the primary cluster.

The request mirroring pattern doesn't impact the traffic sent to the primary cluster, and because Envoy will collect all statistics for the shadow cluster, it's a helpful technique for testing.

In addition to the "fire and forget", make sure that the requests you're mirroring are idempotent. Otherwise, mirrored requests can mess up the backends your services talk to.

The authority/host headers on the shadowed request will have the -shadow string appended.

To configure the mirroring policy, we use the request\_mirror\_policies field on the route for which we want to mirror the traffic. We can specify one or more mirroring policies and the fraction of traffic we want to mirror:

route\_config:  
 name: my\_route  
 virtual\_hosts:  
 - name: httpbin  
 domains: ["\*"]  
 routes:  
 - match:  
 prefix: /  
 route:  
 cluster: httpbin  
 request\_mirror\_policies:  
 cluster: mirror\_httpbin  
 runtime\_fraction:  
 default\_value:  
 numerator: 100  
 ...

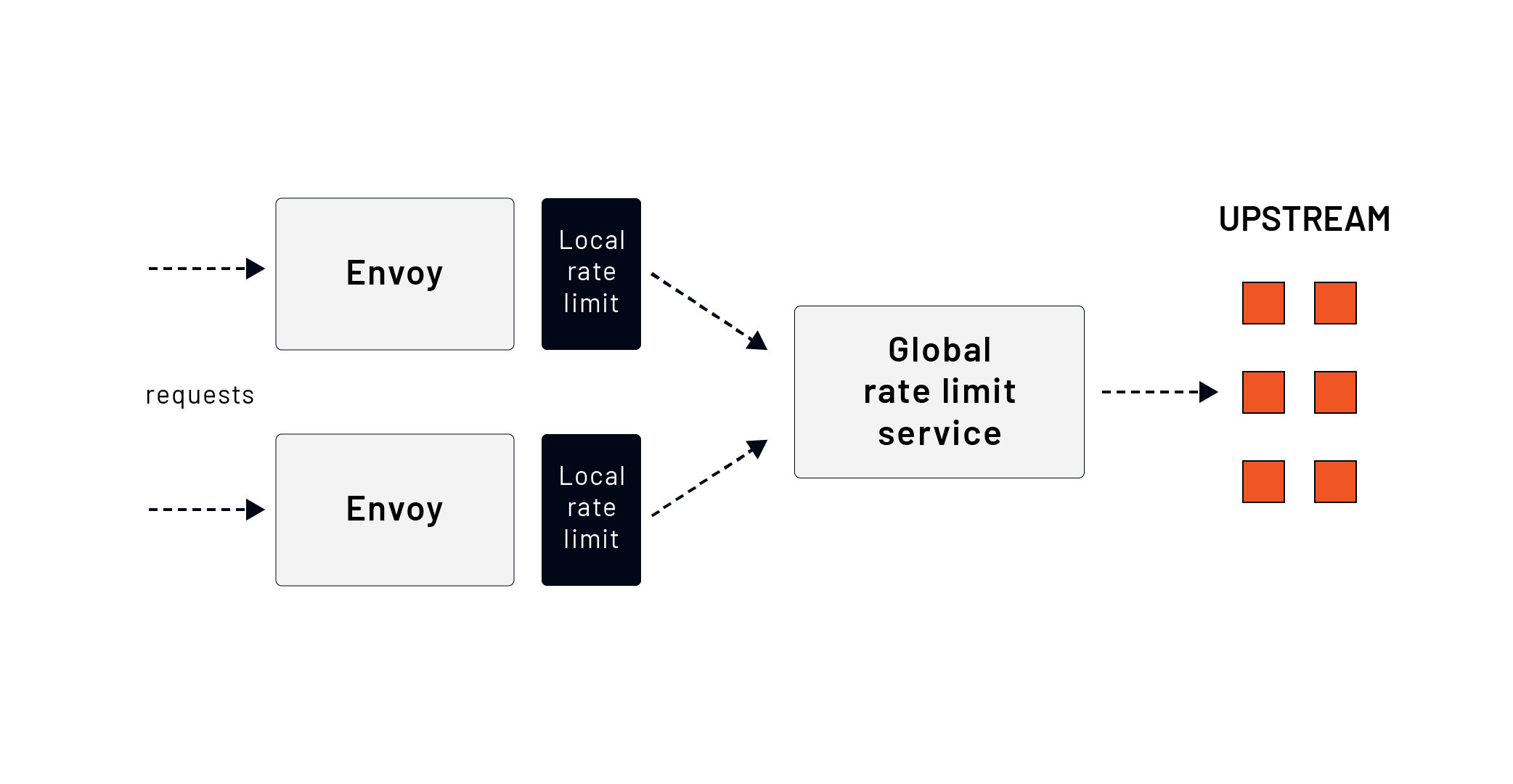
The above configuration will take 100% of incoming requests sent to cluster httpbin and mirror them to mirror\_httpbin.

# 2.11Rate limiting

Rate limiting is a strategy for limiting incoming requests. It specifies how many times a host or a client sends a request within a specific timeframe. Once it reaches the limit, for example, 100 requests per second, we say that the client making the call is rate limited. Any rate limited requests are rejected and never reach the upstream service. Later on, we'll also talk about circuit breakers that can be used and how rate limits can limit the load to upstreams and prevent cascading failures.

Envoy supports rate limiting at the global (distributed) and local (non-distributed) levels.

The difference between global and local rate limiting is that we're trying to control access to a set of upstreams that are **shared** between multiple Envoy instances with the global rate limit. For example, we want to rate limit access to a database called multiple Envoy proxies. On the other hand, the local rate limit applies per each Envoy instance.

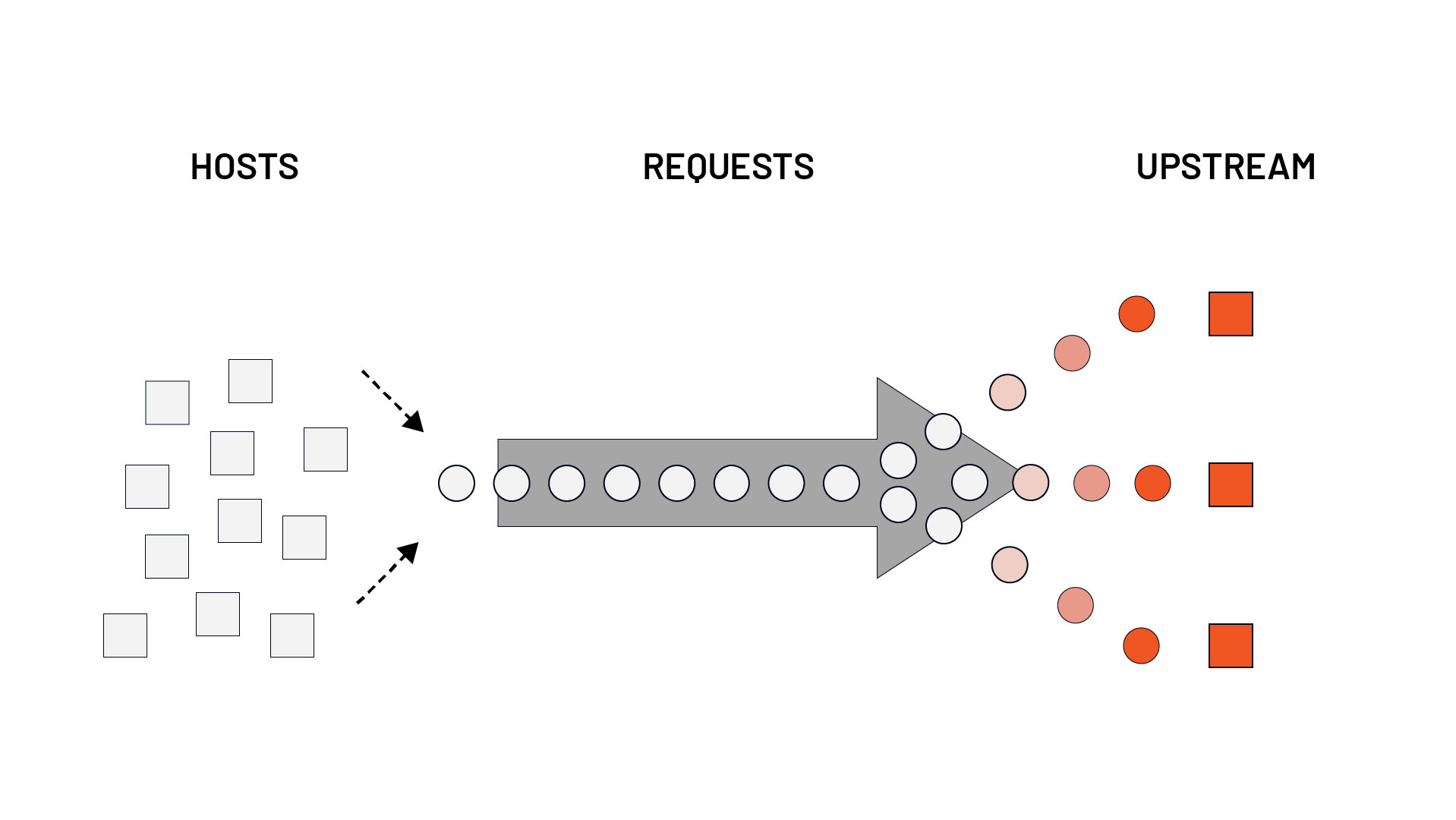


Both local and global rate limits can be used together, and Envoy applies them in two stages. First, the local rate limit is applied, then the global rate limit.

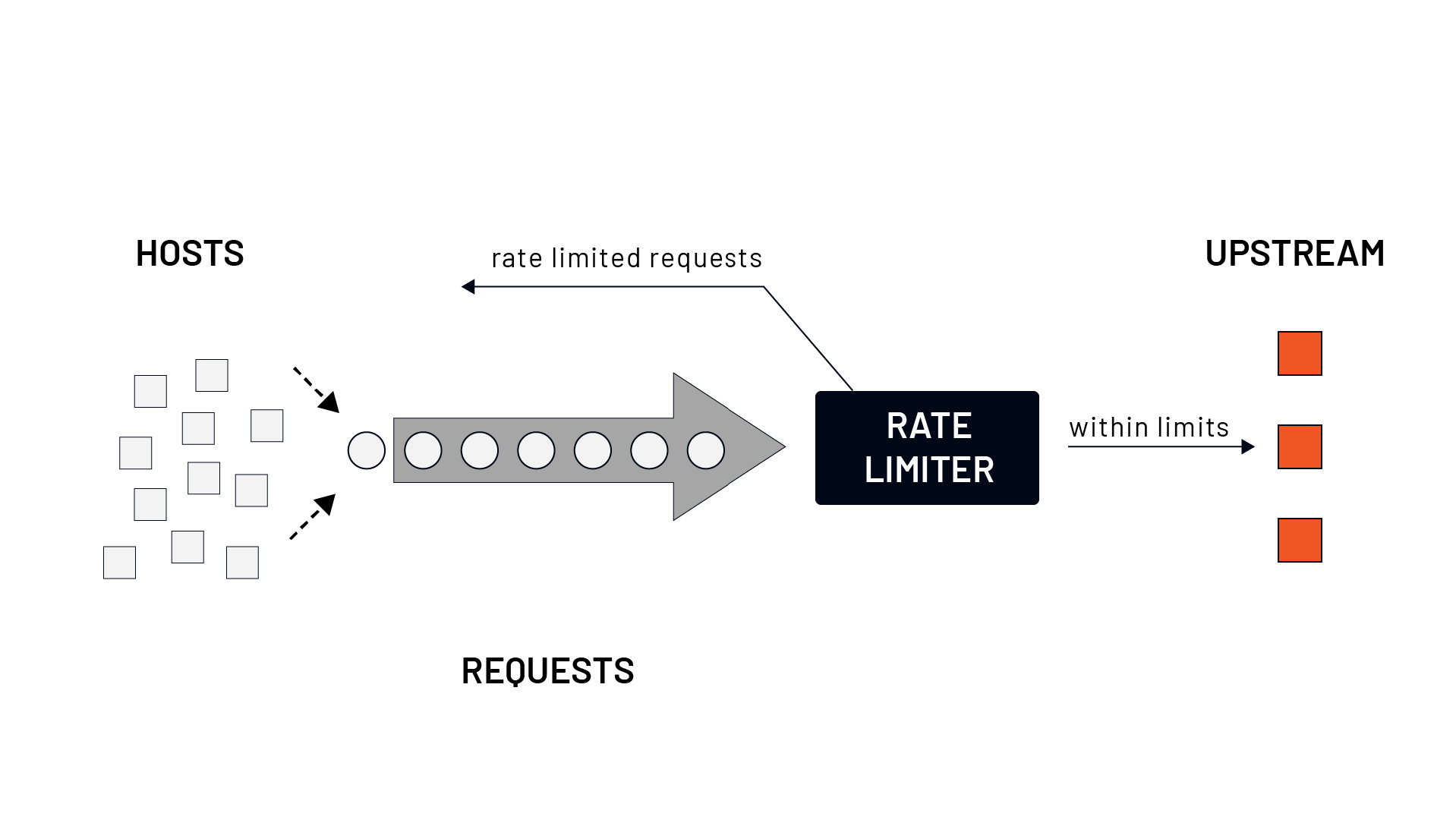
We'll dive deeper into the global and local rate-limiting in the upcoming sections and explain how both scenarios work.

# 2.12 Global rate limiting

Global or distributed rate limiting is useful when many hosts send requests to a small number of upstream servers, and the average latency is low.



The requests become backed up because hosts can't process them quickly enough. In this case, the numerous downstream hosts can overwhelm the small number of upstream hosts. The global rate limiter helps to prevent cascading failures.



Envoy integrates with any external rate-limiting services that implement the defined RPC/IDL protocol. The reference implementation of the service uses Go, gRPC, and Redis for its backend and is available [here](https://github.com/envoyproxy/ratelimit).

The Envoy calls the external rate limit service (that stores the stats and tracks requests in Redis, for example) to get back the answer if the request should be rate-limited or not.

Using an external rate-limiting service, we can apply the limits to a set of services or, if we're talking about service mesh, to all services in the mesh. We can control the number of requests entering the mesh as a whole.

To control the rate of requests at an individual service level, we can use the local rate limiter. The local rate limiter allows us to have individual rate limits for each service. The local and global rate limiting is typically used together.

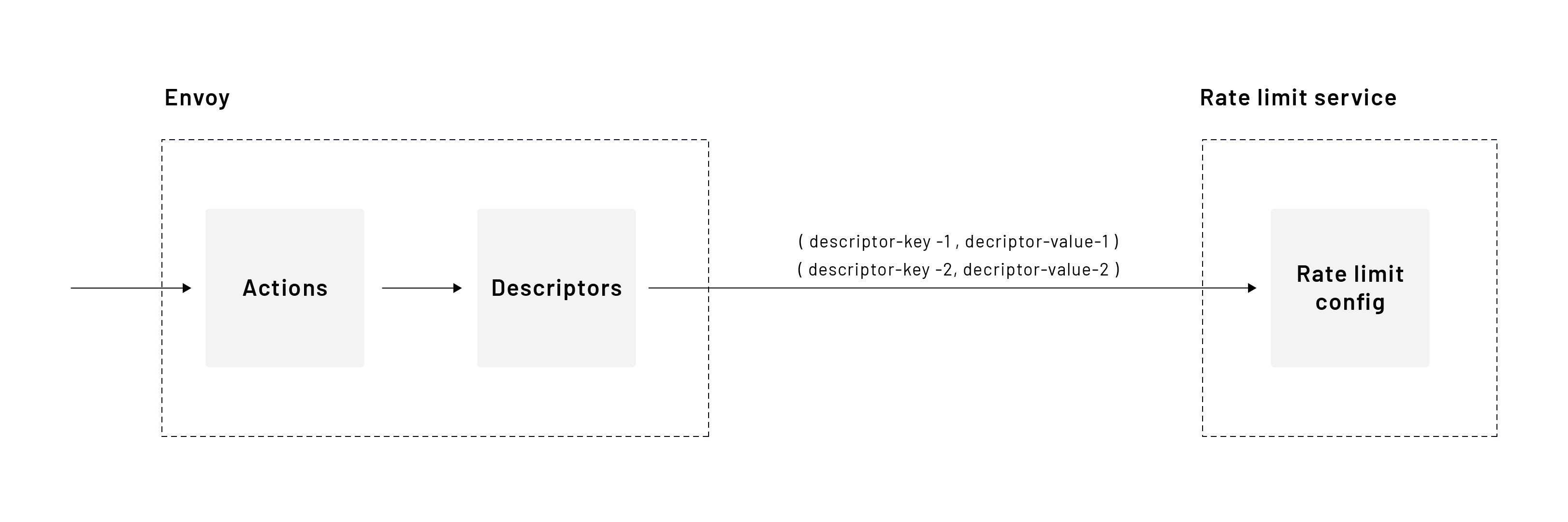
## Configuring global rate limits

When configuring global rate limits, we have to set up two parts -- the client-side (Envoy) and the server-side (rate limit service).

We can configure the rate limit service on the Envoy side either as a **network-level rate limit filter** or an **HTTP-level rate limit filter**.

When using the rate limit filter on the network level, Envoy invokes the rate limit service for every new connection made to the listener where we configured the filter. Similarly, with the HTTP level rate limit filter, Envoy calls the rate limit service for every new request on the listener where the filter is installed **and** where the routing table specifies that the global rate limit service should be called. All requests to the target upstream cluster and requests from the originating cluster to the target cluster can be rate limited.

Before we can configure the rate limit service, we need to explain the concept of **actions**, **descriptors** (key/value pairs), and a **descriptors list**.



In the Envoy configuration on the route or virtual host level, we define a set of **actions**. Each action contains a list of rate limit actions. Let's consider the following example where rate limits are defined at the virtual host level:

rate\_limits:  
- actions:  
 - header\_value\_match:  
 descriptor\_value: get\_request  
 headers:  
 - name: :method  
 prefix\_match: GET  
 - header\_value\_match:  
 descriptor\_value: path  
 headers:  
 - name: :path  
 prefix\_match: /api  
- actions:  
 - header\_value\_match:  
 descriptor\_value: post\_request  
 headers:  
 - name: :method  
 prefix\_match: POST  
- actions:  
 - header\_value\_match:  
 descriptor\_value: get\_request  
 headers:  
 - name: :method  
 prefix\_match: GET

The above snippet defines three separate actions that contain rate limit actions. Envoy will try to match the request against the rate limit actions and generate descriptors sent to the rate limit service. If Envoy cannot match any single rate limit action to the request, no descriptors get created -- i.e., all rate limit actions have to match.

For example, if we receive a GET request to /api, the first action matches both rate limit actions; hence a following descriptor gets created:

("header\_match": "get\_request"), ("header\_match": "path")

The second action isn't going to match. However, the last one will match as well. Therefore Envoy will send the following descriptors to the rate limit service:

("header\_match": "get\_request"), ("header\_match": "path")  
("header\_match": "get\_request")

Let's look at another example of the client-side configuration that satisfies the following requirements:

* POST requests to /users are rate limited to 10 requests per minute
* requests to /users are rate limited to 20 requests per minute
* requests with a header dev: true to /api are rate limited to 10 requests per second
* requests with a header dev: false to /api are rate limited to 5 requests per second
* any other requests to /api are not rate limited

Notice that this time, we're defining the rate limits at the route level.

routes:  
- match:  
 prefix: "/users"  
 route:  
 cluster: some\_cluster  
 rate\_limits:  
 - actions:  
 - generic\_key:  
 descriptor\_value: users  
 - header\_value\_match:  
 descriptor\_value: post\_request  
 headers:  
 - name: ":method"  
 exact\_match: POST  
 - actions:  
 - generic\_key:  
 descriptor\_value: users  
- match:  
 prefix: "/api"  
 route:  
 cluster: some\_cluster  
 rate\_limits:  
 - actions:  
 - generic\_key:  
 descriptor\_value: api  
 - request\_headers:  
 header\_name: dev  
 descriptor\_key: dev\_request  
...  
http\_filters:  
- name: envoy.filters.http.ratelimit  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.ratelimit.v3.RateLimit  
 domain: some\_domain  
 enable\_x\_ratelimit\_headers: DRAFT\_VERSION\_03  
 rate\_limit\_service:  
 transport\_api\_version: V3  
 grpc\_service:  
 envoy\_grpc:  
 cluster\_name: rate-limit-cluster

The above configuration contains the rate\_limits configuration per route and the envoy.filters.http.ratelimit filter configuration. The filter configuration points to the upstream cluster of the rate limit service. We've also set the domain name (domain) and the enabled\_x\_ratelimit\_headers field that specifies we want to use the x-ratelimit headers. The domain name allows us to isolate a set of rate limit configurations per arbitrary domain name.

If we look at the rate limit config in the routes, notice how we split up the actions to match the different rate limits we want to set. For example, we have one action with the api generic key and the request headers. However, within the same configuration, we also have an action with only the generic key set. This allows us to configure different rate limits based on these actions.

Let's translate the actions into descriptors:

GET /users --> ("generic\_key": "users")  
  
POST /users --> ("generic\_key": "users"), ("header\_match": "post\_request")  
  
GET /api   
dev: some\_header\_value --> ("generic\_key": "api"), ("dev\_request": "some\_header\_value")

The difference between the header\_match and request\_headers is that with the latter, we can create rate limits based on the specific header values (for example, dev: true or dev: something as the value of the header becomes part of the descriptor).

On the rate limit service side, we need to develop a configuration that specifies the rate limits based on the descriptors sent by Envoy.

For example, if we send a GET request to /users, Envoy sends the following descriptor to the rate limit service: ("generic\_key": "users"). However, if we send a POST request, the descriptor list looks like this:

("generic\_key": "users"), ("header\_match": "post\_request")

The rate limit service configuration is hierarchical to allow for matching nested descriptors. Let's look at how the rate limit service configuration would look like for the above descriptors:

domain: some\_domain  
descriptors:  
- key: generic\_key  
 value: users  
 rate\_limit:  
 unit: MINUTE  
 requests\_per\_unit: 20  
 descriptors:  
 - key: header\_match  
 value: post\_request  
 rate\_limit:  
 unit: MINUTE  
 requests\_per\_unit: 10  
- key: generic\_key  
 value: api  
 descriptors:  
 - key: dev\_request  
 value: true  
 rate\_limit:  
 unit: SECOND  
 requests\_per\_unit: 10  
 - key: dev\_request  
 value: false  
 rate\_limit:  
 unit: SECOND  
 requests\_per\_unit: 5

We've mentioned the domain value before on the Envoy side configuration. We can now see how to use the domain name. We could use the same descriptor names across the whole fleet of proxies but have them separated by the domain name.

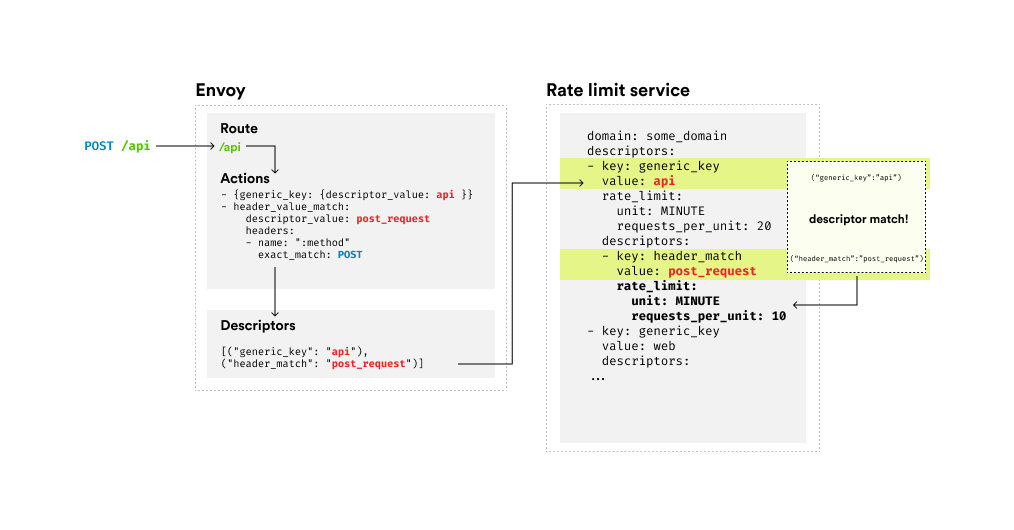
Let's look at how matching on the rate limit service works for different requests:

|  |  |  |  |
| --- | --- | --- | --- |
| Incoming request | Generated descriptor | Rate limit | Explanation |
| GET /users | ("generic\_key": "users") | 20 req/min | The key users matches the first level in the configuration. Since the second level in the config (header\_match) is not included in the descriptor, the rate limit for the users key is used. |
| POST /users | ("generic\_key": "users"), ("header\_match": "post\_request") | 10 req/min | The sent descriptor matches the users as well as the header\_match, so the rate limit under the header\_match descriptor is used. |
| GET /api | ("generic\_key": "api") | No rate limit | We only have a first-level match for api descriptor. However, there are no rate limits configured. For the rate limit to be enforced, we need the second descriptors that are only set if the header dev is present in the incoming request. |
| GET /api dev: true | ("generic\_key": "api"), ("dev\_request": "true") | 10 req/second | The second descriptor in the list matches the second level in the configuration (i.e. we match api and then also dev\_request: true value). |
| GET /api dev: false | ("generic\_key": "api"), ("dev\_request": "false") | 5 req/second | The second descriptor in the list matches the second level in the configuration (i.e. we match api and then also dev\_request: true value). |
| GET /api dev: hello | ("generic\_key": "api"), ("dev\_request": "hello") | No rate limit | The second descriptor in the list doesn't match any of the second-level descriptors in the config. |

In addition to the actions we used in the above example, the table below shows the other actions we can use to create descriptors:

|  |  |
| --- | --- |
| Action name | Description |
| source\_cluster | rate limit on source cluster |
| destination\_cluster | rate limit on destination cluster |
| request\_headers | Rate limit on request headers |
| remote\_address | rate limit on remote address |
| generic\_key | rate limit on a generic key |
| header\_value\_match | rate limit on the existence of request headers |
| metadata | rate limit on metadata |

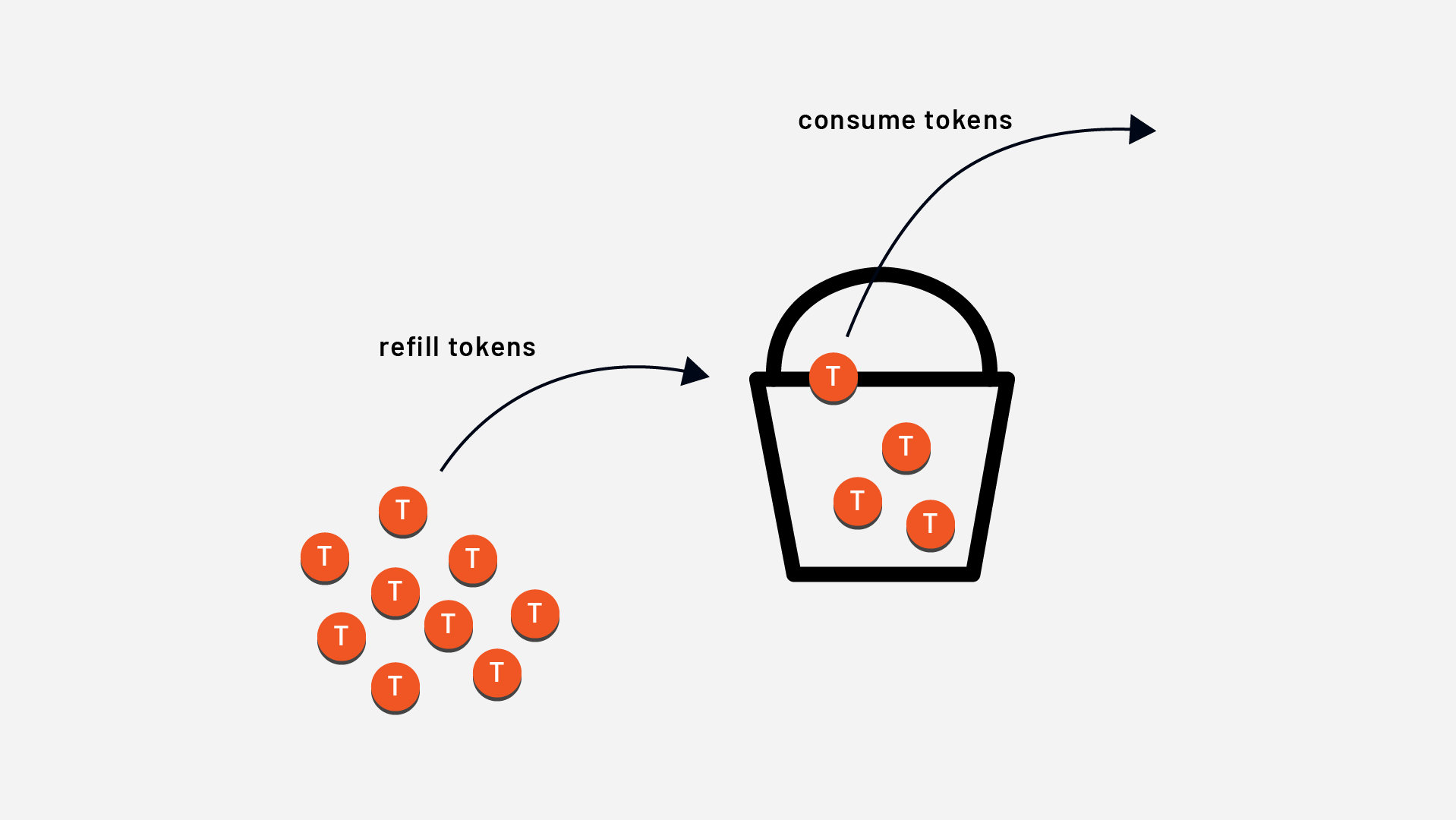
The figure below summarizes how the actions relate to the descriptors and the actual rate limit configuration on the rate limit service.



# 2.13 Local rate limiting

The local rate limit filter applies a **token bucket** rate limit to incoming connections processed by the filter chain.

The basis of the token bucket algorithm is the analogy of tokens in a bucket. The bucket gets refilled with tokens at a fixed rate. Each time a request or connection is received, we check if there are any tokens left in the bucket. If there are, a token gets removed from the bucket, and the request gets processed. If there are no tokens left, the request gets dropped (i.e., rate-limited).



The local rate limiting can be configured globally at the listener level or the virtual host or route level, just like the global rate limit. We can also combine the global and local rate limits in the same configuration.

The token\_bucket specifies the configuration to use for requests processed by the filter. It includes the maximum number of tokens (max\_tokens) bucket can hold, the number of tokens to add during each fill interval (tokens\_per\_refill), and the fill internal (fill\_interval).

Here's an example configuration for a bucket that can hold a maximum of 5000 tokens. Every 30 seconds, 100 tokens get added to the bucket. The bucket will never contain more than 5000 tokens.

token\_bucket:  
 max\_tokens: 5000  
 tokens\_per\_fill: 100  
 fill\_interval: 30s

To control whether the token buckets are shared across all workers (i.e., per Envoy process) or used per-connection basis, we can set the local\_rate\_limit\_per\_downstream\_connection field. The default value is false, which means rate limits get applied per Envoy process.

The two settings controlling if the rate limit is enabled or enforced for a certain fraction of the requests are called filter\_enabled and filter\_enforced. Both of these values are set to 0% by default.

A rate limit can be enabled but not necessarily enforced for a fraction of the requests. For example, we can enable the rate limit for 50% of the requests. Then, within those 50% requests, we can enforce the rate limits.



The following configuration enables and enforces rate limits for all incoming requests:

token\_bucket:  
 max\_tokens: 5000  
 tokens\_per\_fill: 100  
 fill\_interval: 30s  
filter\_enabled:  
 default\_value:  
 numerator: 100  
 denominator: HUNDRED  
filter\_enforced:  
 default\_value:  
 numerator: 100  
 denominator: HUNDRED

We also can add request and response headers for rate limited requests. We can provide a list of headers in the request\_headers\_to\_add\_when\_not\_enforced field, and Envoy will add a request header to each rate-limited request forwarded to upstream. Note that this will only happen when the filter is enabled but not enforced.

For the response headers, we can use response\_headers\_to\_add field. We can provide a list of headers that will get added to the response of the requests that have been rate-limited. This will only happen when filters are either enabled or fully enforced.

If we build on the previous example, here's an example of how to add specific response headers to all rate limited requests:

token\_bucket:  
 max\_tokens: 5000  
 tokens\_per\_fill: 100  
 fill\_interval: 30s  
filter\_enabled:  
 default\_value:  
 numerator: 100  
 denominator: HUNDRED  
filter\_enforced:  
 default\_value:  
 numerator: 100  
 denominator: HUNDRED  
response\_headers\_to\_add:  
 - append: false  
 header:  
 key: x-local-rate-limit  
 value: 'true'

We can configure the local rate limiter such that all virtual hosts and routes share the same token buckets. To enable the local rate limit filter globally (not to be confused with the global rate limit filter), we can provide the configuration for it in the http\_filters list.

For example:

...  
http\_filters:  
- name: envoy.filters.http.local\_ratelimit  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.local\_ratelimit.v3.LocalRateLimit  
 stat\_prefix: http\_local\_rate\_limiter  
 token\_bucket:  
 max\_tokens: 10000  
 ...  
- name: envoy.filters.http.router  
...

If we want to enable the local rate limit per route, we still have to add the filter to the http\_filters list without any configuration. Then, in the route configuration, we can use the typed\_per\_filter\_config and specify the local rate limit filter configuration.

For example:

...  
route\_config:  
 name: my\_route  
 virtual\_hosts:  
 - name: my\_service  
 domains: ["\*"]  
 routes:  
 - match:  
 prefix: /  
 route:  
 cluster: some\_cluster  
 typed\_per\_filter\_config:  
 envoy.filters.http.local\_ratelimit:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.local\_ratelimit.v3.LocalRateLimit  
 token\_bucket:  
 max\_tokens: 10000  
 tokens\_per\_fill: 1000  
 fill\_interval: 1s  
 filter\_enabled:  
 default\_value:  
 numerator: 100  
 denominator: HUNDRED  
 filter\_enforced:  
 default\_value:  
 numerator: 100  
 denominator: HUNDRED  
http\_filters:  
- name: envoy.filters.http.local\_ratelimit  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.local\_ratelimit.v3.LocalRateLimit  
 stat\_prefix: http\_local\_rate\_limiter  
- name: envoy.filters.http.router

The above configuration loads the local rate limit filter in the http\_filter list. We configure it within the route configuration by using typed\_per\_filter\_config and referring to the filter by name envoy.filters.http.local\_ratelimit.

## Using descriptors for local rate limiting

Like we used descriptors when doing global rate limiting, we can also use them for local per-route rate limiting. We need to configure two parts: the actions on the routes and the descriptors list in the local rate limit filter configuration.

We can define actions for the local rate limit in the same way we defined them for the global rate limits:

...  
route\_config:  
 name: my\_route  
 virtual\_hosts:  
 - name: my\_service  
 domains: ["\*"]  
 routes:  
 - match:  
 prefix: /  
 route:  
 cluster: some\_cluster  
 rate\_limits:  
 - actions:  
 - header\_value\_match:  
 descriptor\_value: post\_request  
 headers:  
 - name: ":method"  
 exact\_match: POST  
 - header\_value\_match:  
 descriptor\_value: get\_request  
 headers:  
 - name: ":method"  
 exact\_match: GET  
...

The second part is writing the configuration to match the generated descriptors and provide the token bucket information. This gets done in the rate limit filter configuration under the descriptors field.

For example:

typed\_per\_filter\_config:  
 envoy.filters.http.local\_ratelimit:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.local\_ratelimit.v3.LocalRateLimit  
 stat\_prefix: some\_stat\_prefix  
 token\_bucket:  
 max\_tokens: 1000  
 tokens\_per\_fill: 1000  
 fill\_interval: 60s  
 filter\_enabled:  
 ...  
 filter\_enforced:  
 ...  
 descriptors:  
 - entries:  
 - key: header\_match  
 value: post\_request  
 token\_bucket:  
 max\_tokens: 20  
 tokens\_per\_fill: 5  
 fill\_interval: 30s  
 - entries:  
 - key: header\_match  
 value: get\_request  
 token\_bucket:  
 max\_tokens: 50  
 tokens\_per\_fill: 5  
 fill\_interval: 20s  
...

For all POST requests (i.e. ("header\_match": "post\_request")) the bucket is set to 20 tokens and it refills every 30 seconds with 5 tokens. For all GET requests, the bucket can hold a maximum of 50 tokens and refills every 20 seconds with 5 tokens.

# 2.14 Rate limiting statistics

Whether using global or local rate-limiting, Envoy will emit metrics described in the table below. We can set the stats prefix using the stat\_prefix field when configuring the filters.

Each metric name is prefixed either with <stat\_prefix>.http\_local\_rate\_limit.<metric\_name> when using local rate limiter and cluster.<route\_target\_cluster>.ratelimit.<metric\_name> when using global rate limiter.

|  |  |  |
| --- | --- | --- |
| Rate limiter | Metric name | Description |
| Local | enabled | Total number of requests for which the rate limiter was called |
| Local/Global | ok | Total under limit responses from the token bucket |
| Local | rate\_limited | Total responses without an available token (but not necessarily enforced) |
| Local | enforced | Total number of rate-limited requests (e.g. HTTP 429 is returned) |
| Global | over\_limit | Total over-limit responses from the rate limit service |
| Global | error | Total errors contacting the rate limit service |
| Global | failure\_mode\_allowed | Total requests that were errors but were allowed due to failure\_mode\_deny setting |

# 3.0 Clusters

In this module, we'll learn about the clusters and how to manage them. In the clusters configuration section of Envoy, we can configure features such as load balancing, health checking, connection pooling, outlier detection, and more.

By the end of this module, you'll have an understanding of how clusters and endpoints work and how to configure load balancing strategies, outlier detection, and circuit breaking.

# 3.1 Service discovery

Clusters can be configured statically in the configuration file or dynamically through the cluster discovery service (CDS) API. Each cluster is a collection of endpoints that Envoy needs to resolve to send the traffic to.

The process of resolving the endpoints is known as **service discovery**.

## What are endpoints?

A cluster is a collection of endpoints that identify a specific host. Each endpoint has the following properties:

**Address (address)**

The address represents the upstream host address. The form of the address depends on the cluster type. The address is expected to be an IP for STATIC or EDS cluster types, and for LOGICAL or STRICT DNS cluster types, the address is expected to be a hostname that's resolved via DNS.

**Hostname (hostname)**

A hostname associated with the endpoint. Note that the hostname is not used for routing or resolving addresses. It's associated with the endpoint and can be used for any features requiring hostname, such as auto host rewrite.

**Health check configuration (health\_check\_config)**

The optional health check configuration is used for the health checker to contact the health checked host. The configuration contains the hostname and the port where the host can be contacted to perform the health check. Note that this configuration is applicable only for upstream clusters with active health checking enabled.

## Service discovery types

There are five supported service discovery types -- let's look at them in more detail.

### Static (STATIC)

Static service discovery type is the simplest. In the configuration we specify a resolved network name for each host in the cluster. For example:

clusters:  
 - name: my\_cluster\_name  
 type: STATIC  
 load\_assignment:  
 cluster\_name: my\_service\_name  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 8080

Note that if we don't provide the type, it defaults to STATIC.

### Strict DNS (STRICT\_DNS)

With strict DNS, Envoy continuously and asynchronously resolves the DNS endpoints defined in the cluster. If the DNS query returns multiple IP addresses, Envoy assumes they are part of the cluster and load balances between them. Similarly, if the DNS query returns zero hosts, Envoy assumes the cluster doesn't have any.

A note on health checking -- the health checking is not shared if multiple DNS names resolve to the same IP address. This might cause an unnecessary load on the upstream host because Envoy performs the health check on the same IP address multiple times (across different DNS names).

When respect\_dns\_ttl field is enabled, we can control the continuous resolution of DNS names using the dns\_refresh\_rate. If not specified, the DNS refresh rate defaults to 5000 ms. Another setting (dns\_failure\_refresh\_rate) controls the refresh frequency during failures. If not provided, Envoy uses the dns\_refresh\_rate.

Here's an example of STRICT\_DNS service discovery type:

clusters:  
 - name: my\_cluster\_name  
 type: STRICT\_DNS  
 load\_assignment:  
 cluster\_name: my\_service\_name  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: my-service  
 port\_value: 8080

### Logical DNS (LOGICAL\_DNS)

The logical DNS service discovery is similar to the strict DNS, and it uses the asynchronous resolution mechanism. However, it only uses the first IP address that's returned when a new connection needs to be initiated.

Therefore, a single logical **connection pool** may contain physical connections to a variety of different upstream hosts. These connections never drain, even when the DNS resolution returns zero hosts.

**What is a connection pool?** Each endpoint in the cluster will have one or more connection pools. For example, depending on the supported upstream protocols, there might be one connection pool per protocol allocated. Each worker thread in Envoy also maintains its connection pool for each cluster. For example, if Envoy has two threads and a cluster that supports both HTTP/1 and HTTP/2, there will be at least four connection pools. The way connection pools are is based on the underlying wire protocol. With HTTP/1.1, the connection pool acquires connections to the endpoint as needed (up to the circuit breaking limit). Requests are bound to connections as they become available. When using HTTP/2, the connection pool multiplexes multiple requests over a **single connection**, up to the limits specified by the max\_concurrent\_streams and max\_requests\_per\_connections. The HTTP/2 connection pool establishes as many connections as needed to serve the requests.

A typical use case for logical DNS is for large-scale web services. Typically using round-robin DNS, they return a different result of multiple IP addresses on each query. If we used strict DNS resolution, Envoy would assume cluster endpoints change on each resolution internally and would drain the connection pools. With logical DNS, the connections will stay alive until they get cycled.

Like the strict DNS, the logical DNS also uses the respect\_dns\_ttl and the dns\_refresh\_rate fields to configure the DNS refresh rate.

clusters:  
 - name: my\_cluster\_name  
 type: LOGICAL\_DNS  
 load\_assignment:  
 cluster\_name: my\_service\_name  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: my-service  
 port\_value: 8080

### Endpoints discovery service (EDS)

Envoy can use the endpoint discovery services to fetch the cluster endpoints. Typically, this is the preferred service discovery mechanism. Envoy gets explicit knowledge of each upstream host (i.e., no need to route through a DNS resolved load balancer). Each endpoint can carry extra attributes that can inform Envoy of the load balancing weight and canary status zone, and so on.

clusters:  
 - name: my\_cluster\_name  
 type: EDS  
 eds\_cluster\_config:  
 eds\_config:  
 ...

We explain the dynamic configuration in more detail in the [Dynamic configuration and xDS] chapter.

### Original destination (ORIGINAL\_DST)

We use the original destination cluster type when connections to Envoy go through iptables REDIRECT or TPROXY target or with the Proxy Protocol.

In this scenario, the requests are forwarded to upstream hosts as addressed by the redirection metadata (for example, using the x-envoy-original-dst-host header) without any configuration or upstream host discovery.

The connections to upstream hosts are pooled and flushed when they've been idle longer than specified in the cleanup\_interval field (defaults to 5000 ms).

clusters:  
 - name: original\_dst\_cluster  
 type: ORIGINAL\_DST  
 lb\_policy: ORIGINAL\_DST\_LB

The only load balancing policy that the ORIGINAL\_DST cluster type can use is the ORIGINAL\_DST\_LB policy.

In addition to the above service discovery mechanisms, Envoy also supports custom cluster discovery mechanisms. We can configure a custom discovery mechanism using the cluster\_type field.

Envoy supports two types of health checks, active and passive. We can use both types of health checking at the same time. With active health checking, Envoy periodically sends a request to the endpoints to check its status. With passive health checking, Envoy monitors how the endpoints respond to connections. It enables Envoy to detect an unhealthy endpoint even before the active health check marks it as unhealthy. Passive health checking in Envoy is realized through outlier detection.

# 3.2 Active health checking

Envoy supports different active health check methods on endpoints: HTTP, TCP, gRPC, and Redis health check. The health check method can be configured for each cluster separately. We can configure the health checks with the health\_checks field in the cluster configuration.

Regardless of the selected health check method, a couple of common configuration settings need to be defined.

The **timeout** (timeout) represents the time alloted to wait for a health check response. If the response is not reached within the time value specified in this field, the health check attempt will be considered a failure. The **interval** specifies the time cadence between health checks. For example, an interval of 5 seconds will trigger a health check every 5 seconds.

The other two required settings can be used to determine when a specific endpoint is considered healthy or unhealthy. The healthy\_threshold specifies the number of "healthy" health checks (e.g., HTTP 200 response) required before an endpoint is marked healthy. The unhealthy\_threshold does the same, but with "unhealthy" health checks -- it specifies the number of unhealthy health checks required before an endpoint is marked unhealthy.

1. HTTP health checks

Envoy sends an HTTP request to the endpoint. If the endpoint responds with an HTTP 200, Envoy considers it healthy. The 200 response is the default response that's regarded as a healthy response. Using the expected\_statuses field, we can customize that by providing a range of HTTP statuses considered healthy.

If the endpoint responds with an HTTP 503, the unhealthy\_threshold is ignored, and the endpoint is considered unhealthy immediately.

clusters:  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout: 1s  
 interval: 0.25s  
 unhealthy\_threshold: 5  
 healthy\_threshold: 2  
 http\_health\_check:  
 path: "/health"  
 expected\_statuses:  
 - start: 200  
 end: 299  
 ...

For example, the above snippet defines an HTTP health check where Envoy will send an HTTP request to the /health path to the endpoints in the cluster. Envoy sends the request every 0.25s (interval) and waits for 1s (timeout) before timing out. To be considered healthy, the endpoint must respond with a status between 200 and 299 (expected\_statuses) twice (healthy\_threshold). The endpoint needs to respond with any other status code five times (unhealthy\_threshold) to be considered unhealthy. Additionally, if the endpoint responds with HTTP 503, it's immediately considered unhealthy (the unhealthy\_threshold setting is ignored).

1. TCP health check

We specify a Hex encoded payload (e.g. 68656C6C6F) that gets sent to the endpoint. If we set an empty payload, Envoy will do a connect-only health check where it only attempts to connect to the endpoint and considers it a success if the connection succeeds.

In addition to the payload that gets sent, we also need to specify the responses. Envoy will perform a fuzzy match on the response, and if the response matches the request, the endpoint is considered healthy.

clusters:  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout: 1s  
 interval: 0.25s  
 unhealthy\_threshold: 1  
 healthy\_threshold: 1  
 tcp\_health\_check:  
 send:  
 text: "68656C6C6F"  
 receive:  
 - text: "68656C6C6F"  
 ...

1. gRPC health check

This health check follows the [grpc.health.v1.Health](https://github.com/grpc/grpc/blob/master/src/proto/grpc/health/v1/health.proto) health checking protocol. Check the [GRPC health checking protocol document](https://github.com/grpc/grpc/blob/master/doc/health-checking.md) for more information on how it works.

The two optional configuration values we can set are the service\_name and the authority. The service name is the value set to the service field in the HealthCheckRequest from grpc.health.v1.Health. The authority is the value of the :authority header. If it's empty, Envoy uses the name of the cluster.

clusters:  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout: 1s  
 interval: 0.25s  
 unhealthy\_threshold: 1  
 healthy\_threshold: 1  
 grpc\_health\_check: {}  
 ...

1. Redis health check

The Redis health check sends a Redis PING command to the endpoint and expects a PONG response. If the upstream Redis endpoint responds with anything other than PONG, it immediately causes the health check to fail. We can also specify a key, and Envoy will perform an EXIST <key> command instead of the PING command. The endpoint is healthy if the return value from Redis is 0 (i.e., the key doesn't exist). Any other response is considered a failure.

clusters:  
 - name: my\_cluster\_name  
 health\_checks:  
 - timeout: 1s  
 interval: 0.25s  
 unhealthy\_threshold: 1  
 healthy\_threshold: 1  
 redis\_health\_check:  
 key: "maintenance"  
 ...

The above example checks for the key "maintenance" (e.g. EXIST maintenance), and if the key doesn't exist, the health check passes.

## HTTP health checking filter

The HTTP health checking filter can be used to limit the amount of health checking traffic that gets generated. The filter can run in different modes of operation that control how and whether the traffic is passed to the local service or not (i.e., no pass-through or pass-through).

1. Non-pass-through mode

When running in the non-pass-through mode, the health check request is never sent to the local service. Envoy responds either with HTTP 200 or HTTP 503, depending on the current draining state of the server.

A variation of the non-pass through mode is where the HTTP 200 is returned if at least a specified percentage of endpoints are available in the upstream cluster. The percentage of endpoints can be configured with the cluster\_min\_healthy\_percentages field:

...  
 pass\_through\_mode: false  
 cluster\_min\_healthy\_percentages:  
 value: 15  
...

1. Pass-through mode

In the pass-through mode, Envoy passes every health check request to the local service. The service can either respond with an HTTP 200 or HTTP 503.

An additional setting for the pass-through mode is to use caching. Envoy passes the health check request to the service and caches the result for a period of time (cache\_time). Any subsequent health check requests will use the cached up value. Once the cache is invalidated, the next health check request gets passed to the service again.

...  
 pass\_through\_mode: true  
 cache\_time: 5m  
...

The above snippet enables the pass-through mode with a cache that expires in 5 minutes.

# 3.3 Outlier detection

The second type of health checking is called **passive health checking**. The process called **outlier detection** is a form of passive health checking. It's "passive" because Envoy isn't "actively" sending any requests to determine the health of the endpoints. Instead, Envoy observes the performance of different endpoints to determine if they are healthy or not. If the endpoints are deemed unhealthy, they are removed or ejected from the healthy load balancing pool.

The endpoints' performance is determined through consecutive failures, temporal success rate, latency, and so on.

For outlier detection to work, we need filters to report the errors, timeouts, and resets. Currently, four filters support outlier detection: HTTP router, TCP proxy, Redis proxy, and Thrift proxy.

Detected errors fall in two categories, based on the point of origin:

1. Externally originated errors

These errors are transaction-specific and occur on the upstream server in response to the received request. The errors are generated on the upstream host after Envoy has connected to it successfully. E.g., the endpoint responds with HTTP 500.

1. Locally originated errors

Envoy generates these errors in response to an event that interrupted or prevented communication with the upstream host, e.g., timeouts, TCP resets, inability to connect to a specified port, and so on.

The errors also depend on the filter type. For example, the HTTP router filter can detect two kinds of errors. In contrast, the TCP proxy filter doesn't understand any protocol above the TCP layer and only reports the locally originated errors.

In the configuration, we can specify whether we can distinguish between local and externally generated errors (using split\_external\_local\_origin\_errors field). This allows us to track the errors by separate counters and configure the outlier detection to react to locally generated errors and ignore the externally generated errors, and vice-versa. The default mode is for the errors not to be split (i.e. split\_external\_local\_origin\_errors is false).

## Endpoint ejection

When an endpoint is determined to be an outlier, Envoy will check if it needs to be ejected from the healthy load balancing pool. If no endpoints were ejected, Envoy ejects the outlier (unhealthy) endpoint right away. Otherwise, it checks the max\_ejection\_percent setting to make sure the number of ejected endpoints is below the configured threshold. The endpoint won't be ejected if more than max\_ejection\_percent hosts were already ejected.

Each endpoint gets ejected for a predetermined amount of time. We can configure the ejection time using the base\_ejection\_time value. This value is multiplied by the number of times the endpoint has been ejected in a row. If the endpoints continue to fail, they'll get ejected for longer and longer periods. The second set here is called max\_ejection\_time . It controls the maximum time endpoints get ejected for - i.e., the maximum time endpoints will ever get ejected is specified in the max\_ejection\_time value.

Envoy checks the health of each endpoint at an interval specified in the interval field. For every check the endpoint is healthy, the ejection multiplier gets decremented. After the ejection time passes, the endpoint gets automatically returned to the healthy load balancing pool.

Now that we understand the basics of outlier detection and endpoint ejection, let's look at the different outlier detection methods.

## Detection types

Envoy supports the following five outlier detection types:

1. Consecutive 5xx

This detection type takes into account all generated errors. Envoy internally maps any errors generated by non-HTTP filters to HTTP 5xx codes.

When error types are split, this detection type only counts the externally originated errors and ignores the locally originated errors. If the endpoint is an HTTP server, only 5xx types of errors are taken into account.

If an endpoint returns a certain number of 5xx errors, the endpoint gets ejected. The consecutive\_5xx value controls the number of consecutive 5xx errors.

clusters:  
 - name: my\_cluster\_name  
 outlier\_detection:  
 interval: 5s  
 base\_ejection\_time: 15s  
 max\_ejection\_time: 50s  
 max\_ejection\_percent: 30  
 consecutive\_5xx: 10  
 ...

The above outlier detection will eject a failing endpoint once it fails ten times. The failing endpoint gets ejected for 15 seconds (base\_ejection\_time). The maximum time a single endpoint gets ejected for in the case of multiple ejections is 50 seconds (max\_ejection\_time). Before a failing endpoint gets ejected, Envoy checks if more than 30% of endpoints were already ejected (max\_ejection\_percent) and decides whether to eject the failing endpoint or not.

1. Consecutive gateway failure

The consecutive gateway failure type is similar to the consecutive 5xx type. It considers a subset of 5xx errors, called "gateway errors" (e.g., 502, 503, or 504 status codes) and local origin failures such as timeouts, TCP reset, etc.

This detection type considers the gateway errors in the split mode and is supported only by the HTTP filter. The number of consecutive errors is configurable with the consecutive\_gateway\_failure field.

clusters:  
 - name: my\_cluster\_name  
 outlier\_detection:  
 interval: 5s  
 base\_ejection\_time: 15s  
 max\_ejection\_time: 50s  
 max\_ejection\_percent: 30  
 consecutive\_gateway\_failure: 10  
 ...

1. Consecutive local origin failure

This type is only enabled in the split mode (split\_external\_local\_origin\_errors is true), and it takes into account only the locally originated errors. The number of consecutive failures is configurable via the consecutive\_local\_origin\_failure field. If not provided, it defaults to 5.

clusters:  
 - name: my\_cluster\_name  
 outlier\_detection:  
 interval: 5s  
 base\_ejection\_time: 15s  
 max\_ejection\_time: 50s  
 max\_ejection\_percent: 30  
 consecutive\_local\_origin\_failure: 10  
 ...

1. Success rate

The success rate outlier detection aggregates success rate data from every endpoint in the cluster. Based on the success rate, it then ejects the endpoints at given intervals. All errors are considered in the default mode, while the external and locally originated errors get treated separately in the split mode.

With the success\_rate\_request\_volume value, we can set the minimum request volume. If the request volume is less than the one specified in the field, the success rate for the host will not be calculated. Similarly, we can use the success\_rate\_minimum\_hosts to set the number of endpoints with the minimum required request volume. If the number of endpoints with minimum required request volume is less than the value set in success\_rate\_minimum\_hosts, Envoy will not perform the outlier detection.

The success\_rate\_stdev\_factor is used to determine the ejection threshold. The ejection threshold is the difference between the mean success rate, and the product of this factor and the standard deviation of the mean success rate:

mean - (stdev \* success\_rate\_stdev\_factor)

This factor is divided by a thousand to get a double. That is, if the desired factor is 1.9, the runtime value should be 1900.

1. Failure percentage

The failure percentage outlier detection is similar to the success rate. The difference is that it doesn't rely on the mean success rate of the cluster as a whole. Instead, it compares the value to a user-configured threshold in the failure\_percentage\_threshold field. If the failure percentage of a give host is greater or equal to this value, the host gets ejected.

The minimum hosts and request volume can be configured using the failure\_percentage\_minimum\_hosts and failure\_percentage\_request\_volume.

# 3.4 Circuit breakers

Circuit breaking is an important pattern that can help with service resiliency. The circuit breaker pattern prevents additional failures by controlling and managing access to the failing services. It allows us to fail quickly and apply back pressure downstream as soon as possible.

Let's look at a snippet that defines the circuit breaking:

...  
 clusters:  
 - name: my\_cluster\_name  
 ...  
 circuit\_breakers:  
 thresholds:  
 - priority: DEFAULT  
 max\_connections: 1000  
 - priority: HIGH  
 max\_requests: 2000  
...

We can configure the circuit breaker thresholds for each route priority separately. For example, the higher priority routes should have higher thresholds than the default priority. If any thresholds are exceeded, the circuit breaker trips and the downstream host receives the HTTP 503 response.

There are multiple options we can configure the circuit breakers with:

1. Maximum connections (max\_connections)

Specifies the maximum number of connections that Envoy will make to all endpoints in the cluster. If this number is exceeded, the circuit breaker trips and increments the upstream\_cx\_overflow metric for the cluster. The default value is 1024.

1. Maximum pending requests (max\_pending\_requests)

Specifies the maximum number of requests that get queued while waiting for a ready connection pool connection. When the threshold is exceeded, Envoy increments the stat upstream\_rq\_pending\_overflow for the cluster. The default value is 1024.

1. Maximum requests (max\_requests)

Specifies the maximum number of parallel requests that Envoy makes to all endpoints in the cluster. The default value is 1024.

1. Maximum retries (max\_retries)

Specifies the maximum number of parallel retries that Envoy allows to all endpoints in the cluster. The default value is 3. If this circuit breaker overflows, the upstream\_rq\_retry\_overflow counter is incremented.

Optionally, we can combine the circuit breakers with a retry budget (retry\_budget). Specifying a retry budget, we can limit the concurrent retries to the number of active requests.

# 3.5 Load balancing

Load balancing is a way of distributing traffic between multiple endpoints in a single upstream cluster. The reason for distributing traffic across numerous endpoints is to make the best use of the available resources.

To achieve the most efficient use of resources, Envoy provides different load balancing strategies that can be separated into two groups: **global load balancing** and **distributed load balancing**. The difference is that we use a single control plane that decides traffic distribution between the endpoints in the global load balancing. Envoy determines how the load gets distributed (e.g., using active health checking, zone-aware routing, load balancing policy).

One of the techniques for distributing load among multiple endpoints is called **consistent hashing**. The server uses a part of the request to create a hash value to select an endpoint. In modulo hashing, the hash is considered to be a huge number. To get the endpoint index to send the request to, we take the hash modulo the number of available endpoints (index=hash % endpointCount). This approach works well if the number of endpoints is stable. However, if the endpoints are added or removed (i.e., they are unhealthy, we scale them up or down, etc.), most requests will end up on a different endpoint than before.

Consistent hashing is an approach where each endpoint gets assigned multiple has values based on some property. Then, each request gets assigned to the endpoint that has the nearest hash value. The value of this approach is that when we add or remove endpoints, most requests will end up on the same endpoint they did before. Having this "stickiness" is helpful because it won't disturb any caches the endpoints hold.

## Load balancing policy

Envoy uses one of the load balancing policies to select an endpoint to send the traffic to. The load balancing policy is configurable and can be specified for each upstream cluster separately. Note that the load balancing is only performed across healthy endpoints. If there's no active or passive health checking defined, all endpoints are assumed to be healthy.

We can configure the load balancing policy using the lb\_policy field and other fields specific to the selected policy.

### Weighted round robin (default)

Weighted round-robin (ROUND\_ROBIN) selects the endpoint in round-robin order. If endpoints are weighted, then a weighted round-robin schedule is used. This strategy gives us a predictable distribution of requests across all endpoints. The higher weighted endpoints will appear more often in the rotation to achieve effective weighting.

### Weighted least request

The weighted least request (LEAST\_REQUEST) algorithm depends on the weights assigned to endpoints.

**If all endpoint weights are equal**, the algorithm selects N random available endpoints (choice\_count) and picks the one with the fewest active requests.

**If endpoint weights are not equal**, the algorithm shifts into a mode where it uses a weighted round-robin schedule in which weights are dynamically adjusted based on the endpoint's request load at the time of selection.

The following formula is used to calculate the weights dynamically:

weight = load\_balancing\_weight / (active\_requests + 1)^active\_request\_bias

The active\_request\_bias is configurable (defaults to 1.0). The larger the active request bias, the more aggressively active requests will lower the effective weight.

If active\_request\_bias is set to 0, the algorithm behaves like the round-robin and ignores the active request count at the time of picking.

We can set the optional configuration for the weighted least request using the least\_request\_lb\_config field:

...  
 lb\_policy: LEAST\_REQUEST  
 least\_request\_lb\_config:  
 choice\_count: 5  
 active\_request\_bias: 0.5  
...

### Ring hash

The ring hash (or modulo hash) algorithm (RING\_HASH) implements consistent hashing to endpoints. Each endpoint address (default setting) is hashed and mapped onto a ring. Envoy routes the requests to an endpoint by hashing some request property and finding the nearest corresponding endpoint clockwise around the ring. The hash key defaults to the endpoint address; however, it can be changed to any other property using the hash\_key field.

We can configure the ring hash algorithm by specifying the minimum (minimum\_ring\_size) and maximum ring size (maximum\_ring\_size) and use the stats (min\_hashes\_per\_host and max\_hashes\_per\_host) to ensure good distribution. The larger the ring, the better the request distribution will reflect the desired weights. The minimum ring size defaults to 1024 entries (limited to 8M entries), while the maximum ring size defaults to 8M (limited to 8M).

We can set the optional configuration for the ring hash can be set using the ring\_hash\_lb\_config field:

...  
 lb\_policy: RING\_HASH  
 ring\_hash\_lb\_config:  
 minimum\_ring\_size: 2000  
 maximum\_ring\_size: 10000  
...

### Maglev

Like the ring hash algorithm, the maglev (MAGLEV) algorithm also implements consistent hashing to endpoints. The algorithm produces a lookup table that allows finding an item within a constant time. Maglev was designed to be faster than the ring hash algorithm for the lookups and use less memory. You can read more about it in the article [Maglev: A Fast and Reliable Software Network Load Balancer](https://dgryski.medium.com/consistent-hashing-algorithmic-tradeoffs-ef6b8e2fcae8).

We can set the optional configuration for the maglev algorithm using the maglev\_lb\_config field:

...  
 lb\_policy: MAGLEV  
 maglev\_lb\_config:  
 table\_size: 69997  
...

The default table size is 65537, but it can be set to any prime number as long as it's not greater than 5000011.

### Original destination

The original destination is a special purpose load balancer that can only be used with an original destination cluster. We've mentioned the original destination load balancer when we talked about the original destination cluster type.

### Random

As the name suggests, the random (RANDOM) algorithm picks an available random endpoint. If you don't have an active health checking policy configured, the random algorithm performs better than the round-robin.

# 4.0 Dynamic configuration overview

In this module, we'll learn about configuring Envoy proxy using dynamic configuration. Up until now, we've been using static configuration. This module will teach us how to provide configuration for individual resources from the file system or over the network using discovery services at runtime.

By the end of this module, you'll understand the difference between static and dynamic configuration.

# 4.1 Dynamic configuration

One of the powerful features of Envoy is the support for dynamic configuration. Up until now, we've been using static configuration. We've specified the listeners, clusters, routes, and other resources as static resources using the static\_resources field.

When using the dynamic configuration, we don't need to restart the Envoy process to take effect. Instead, Envoy dynamically reloads the configuration by reading it from files on the disk or over the network. The dynamic configuration uses so-called **discovery service APIs** that point to specific parts of the configuration. Collectively, these APIs are also referred to as **xDS**. When using xDS, Envoy calls out to external gRPC/REST-based configuration providers that implement the discovery service APIs to retrieve the configuration.

The external gRPC/REST-based configuration provider is also called a **control plane**. When using files on disk, we don't need a control plane. Envoy ships with a Golang implementation of the control plane, but Java and other implementations of control plane are also used.

There are multiple discovery service APIs within Envoy. All of them are described in the table below.

|  |  |
| --- | --- |
| Discovery service name | Description |
| Listener discovery service (LDS) | Using LDS, Envoy can discover listeners at runtime, including all filter stacks, HTTP filters, and references to RDS. |
| Extension config discovery service (ECDS) | Using ECDS, Envoy can fetch extension configuration (e.g., HTTP filter configuration) independently from the listener. |
| Route discovery service (RDS) | Using RDS, Envoy can discover the entire route configuration for an HTTP connection manager filter at runtime. In combination with EDS and CDS, we can implement complex routing topologies. |
| Virtual host discovery service (VHDS) | Using VHDS allows Envoy to request virtual hosts separately from the route configuration. This is used when there is a large number of virtual hosts in a route configuration. |
| Scoped route discovery service (SRDS) | Using SRDS, we can break up the routing table into multiple pieces. This API is used when we have large route tables. |
| Cluster discovery service (CDS) | Using CDS, Envoy can discover upstream clusters. Envoy will add, update, or remove the clusters gracefully by draining and reconnecting all existing connection pools. Envoy doesn't have to be aware of all clusters at initialization time, as we can configure them later using CDS. |
| Endpoint discovery service (EDS) | Using EDS, Envoy can discover members of an upstream cluster. |
| Secret discovery service (SDS) | Using SDS, Envoy can discover secrets (certificates and private keys, TLS session ticket keys) for its listeners and configuration for peer certificate validation logic. |
| Runtime discovery service (RTDS) | Using RTDS, Envoy can discover runtime layers dynamically. |

**Aggregated discovery service (ADS)**

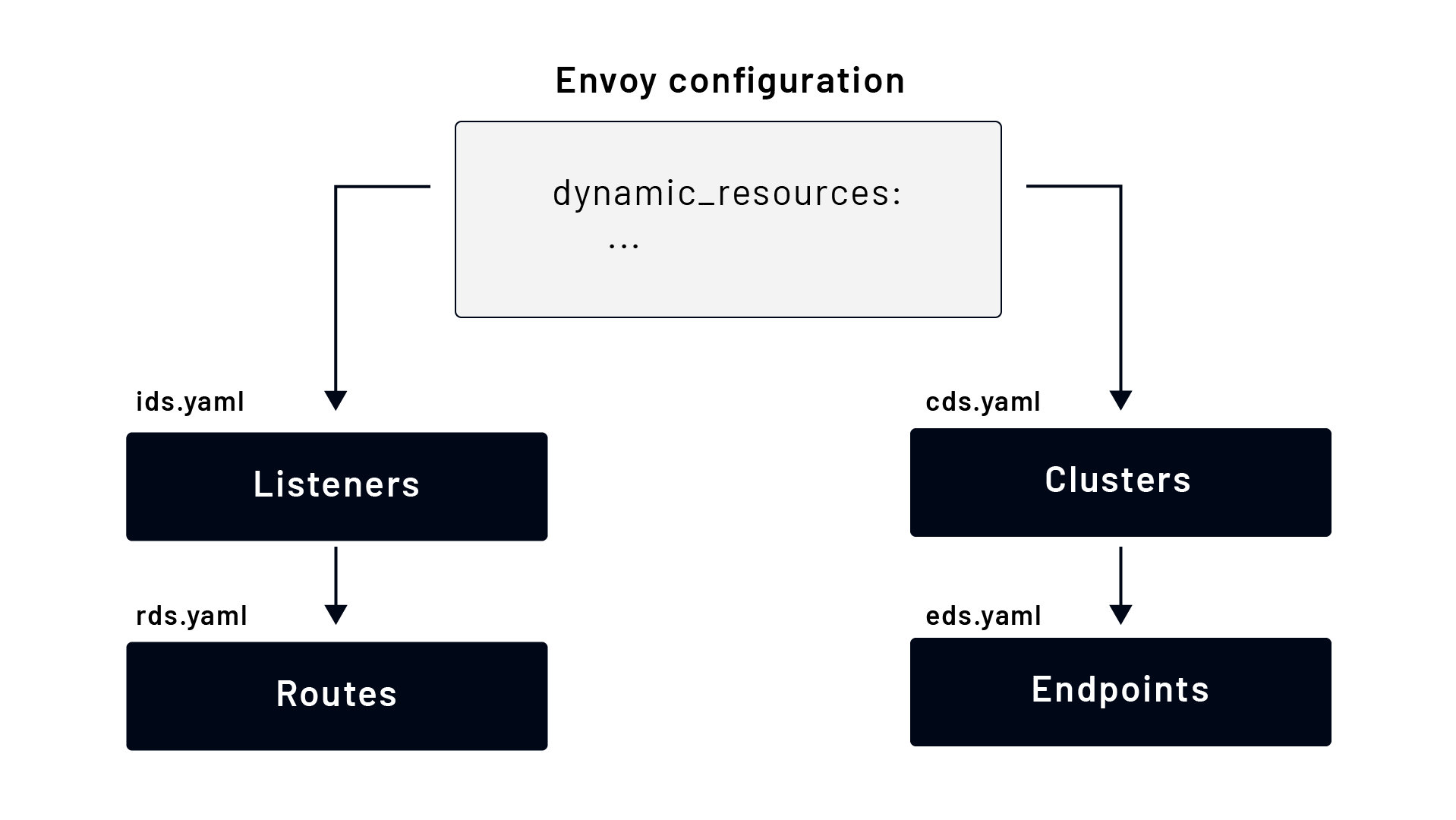
The discovery services in the table are separate and have different gRPC/REST service names. Using the aggregated discovery service (ADS), we can use a single gRPC service that supports all resource types (listeners, routes, clusters, ...) in a single gRPC stream. ADS also ensures the correct sequencing of updates for different resources. Note that ADS only supports gRPC. Without ADS, we'd need to coordinate other gRPC streams to achieve the correct sequence of updates.

**Delta gRPC xDS**

Every time we send a resource update, we have to include all resources. For example, every RDS update must contain every route. If we don't include a route, Envoy considers the route deleted. Doing updates this way results in high bandwidth and computationl costs, especially when there are a lot of resources that are being sent over the network. Envoy supports a delta variant of xDS where we can include only resources we want to add/remove/update to improve on this scenario.

# 4.2 Dynamic configurations from the file system

One way of dynamically providing configuration is by pointing to files on a file system. For dynamic configuration to work, we need to provide the information under the node field. The node field is used to identify a specific Envoy instance if we might have multiple Envoy proxies pointing to the same configuration files.



To point to the dynamic resource, we can use the dynamic\_resources field to tell Envoy where to find dynamic configuration for specific resources. For example:

node:  
 cluster: my-cluster  
 id: some-id  
  
dynamic\_resources:  
 lds\_config:  
 path: /etc/envoy/lds.yaml  
 cds\_config:  
 path: /etc/envoy/cds.yaml

The above snippet is a valid Envoy configuration. The individual configurations for LDS and CDS would be very similar if we'd provide them as static resources. The one difference is that we have to specify the resource type and version information. Here's a snippet of the CDS configuration:

version\_info: "0"  
resources:  
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster  
 name: instance\_1  
 connect\_timeout: 5s  
 load\_assignment:  
 cluster\_name: instance\_1  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 3030

If we wanted to use EDS to provide endpoints for the cluster, we could write the above configuration like this:

version\_info: "0"  
resources:  
- "@type": type.googleapis.com/envoy.config.cluster.v3.Cluster  
 name: instance\_1  
 type: EDS  
 eds\_cluster\_config:  
 eds\_config:  
 path: /etc/envoy/eds.yaml

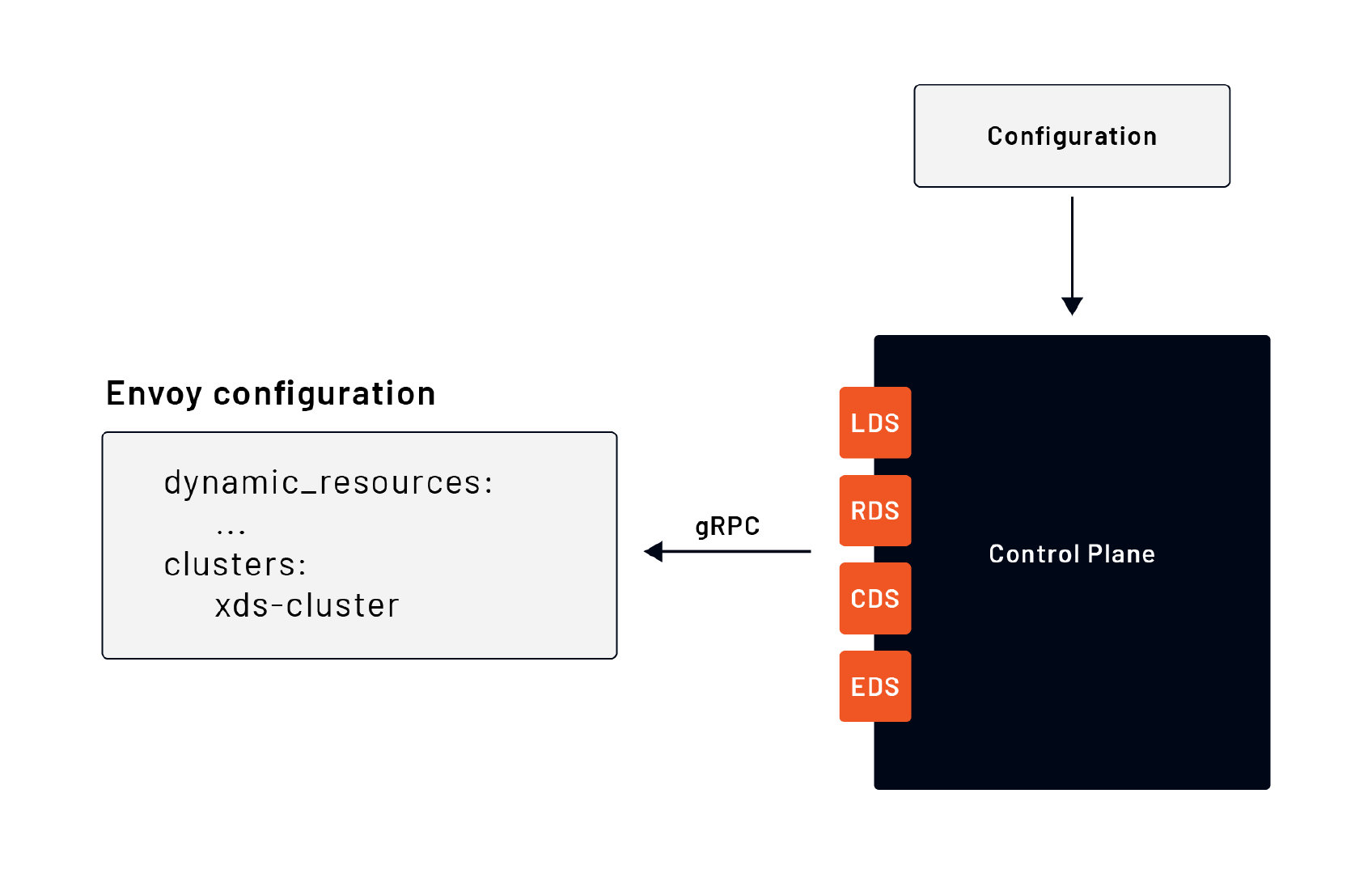
Also, notice we've set the type of the cluster to EDS. The EDS configuration would look like this:

version\_info: "0"  
resources:  
- "@type": type.googleapis.com/envoy.config.endpoint.v3.ClusterLoadAssignment  
 cluster\_name: instance\_1  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 3030

As any of the files get updated, Envoy will automatically reload the configuration. Envoy will output the error if the configuration is invalid but will keep the existing (working) configuration running.

# 4.3 Dynamic configurations from the control plane

Using the control plane to update Envoy is more complex than using configuration from the file system. We have to create our control plane that implements the discovery service interfaces. A simple example of an xDS server implementation is available [here](https://github.com/envoyproxy/go-control-plane/tree/main/internal/example). The example shows how to implement different discovery services and run an instance of the gRPC server Envoy connects to retrieve the configuration.



The dynamic configuration on the Envoy side is similar to the one used for the file system. This time, the difference is that we provide the location of the gRPC server that implements the discovery services. We do that by specifying a cluster through static resources:

...  
dynamic\_resources:  
 lds\_config:  
 resource\_api\_version: V3  
 api\_config\_source:  
 api\_type: GRPC  
 transport\_api\_version: V3  
 grpc\_services:  
 - envoy\_grpc:  
 cluster\_name: xds\_cluster  
 cds\_config:  
 resource\_api\_version: V3  
 api\_config\_source:  
 api\_type: GRPC  
 transport\_api\_version: V3  
 grpc\_services:  
 - envoy\_grpc:  
 cluster\_name: xds\_cluster  
  
static\_resources:  
 clusters:  
 - name: xds\_cluster  
 type: STATIC  
 load\_assignment:  
 cluster\_name: xds\_cluster  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 9090

The control plane doesn't need to operate on Envoy concepts. It may abstract configurations. It can also collect input from users using graphical UI or different YAML, XML, or any other configuration files. The important portion is that regardless of how the high-level configuration comes into the control plane, it needs to be translated into Envoy xDS API.

For example, Istio is a control plane for a fleet of Envoy proxies that can be configured through various custom resource definitions (VirtualService, Gateway, DestinationRule, ...). In addition to the high-level configuration, in [Istio](https://istio.io), the Kubernetes environment and services running inside the cluster are also used as an input to generate the Envoy configuration. Together, the high-level configuration and the discovered services in the environment can be used as input to your control plane. The control plane can take those inputs, transform them into Envoy-readable configuration and send them over gRPC to Envoy instances.

# 5.0 Listener subsystem

In the Listener subsystem module we'll learn about the listener subsystem of the Envoy proxy. We'll introduce the filters, filter chain matching, and different listener filters.

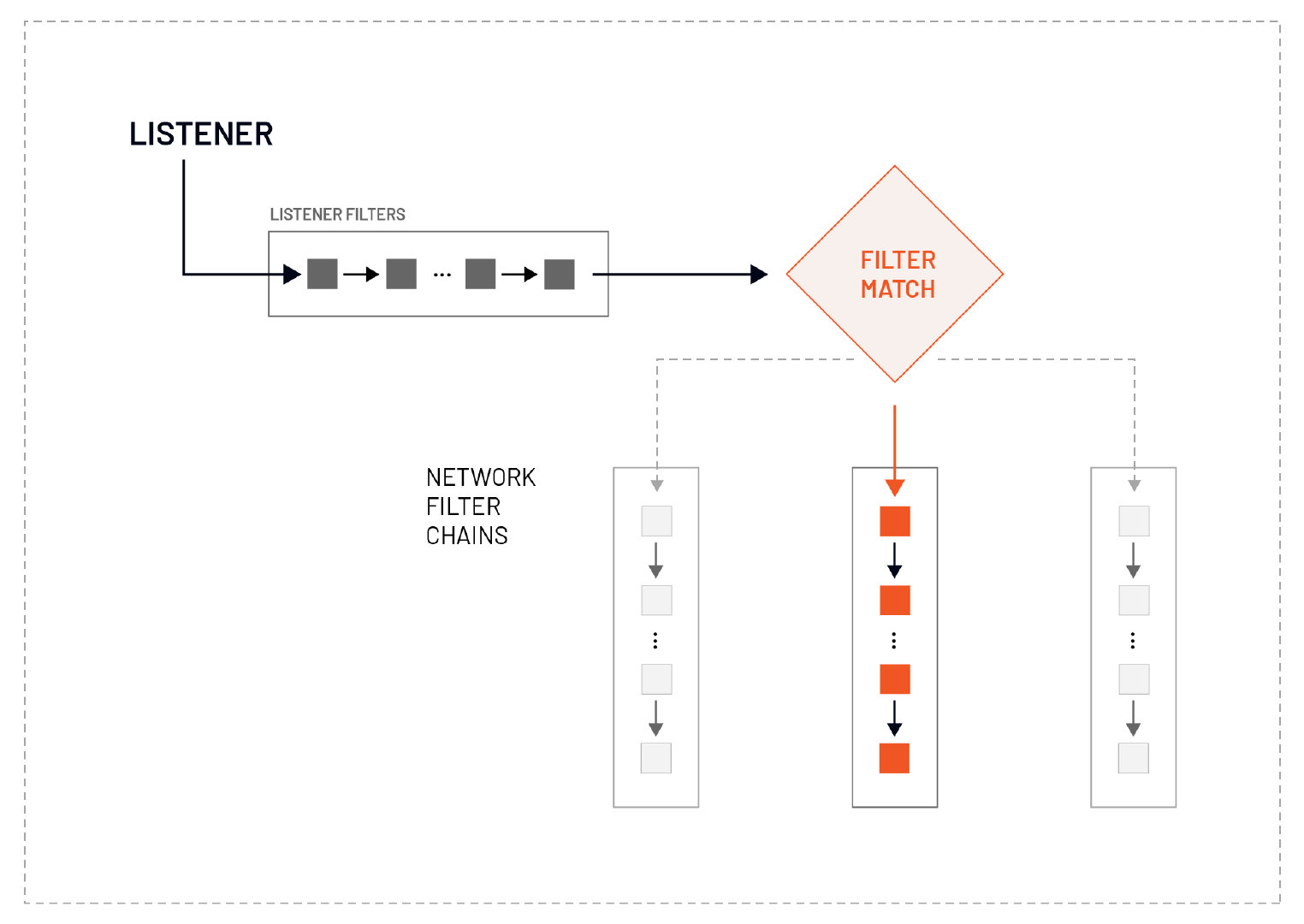
By the end of this module you'll understand the different parts of the listener subsystem and know how filters and filter chain matching works.

# 5.1 Listener filters

As mentioned in the introduction chapter, the **listener subsystem** handles the **downstream** or incoming request processing. The listener subsystem is responsible for incoming requests and the response path to the client. In addition to defining which addresses and ports Envoy "listens" on for incoming requests, we can optionally configure each listener with **listener filters**.

Let's not confuse the listener filters with the network filter chains and the L3/L4 filters we discussed earlier. Envoy processes the listener filters before it processes the network-level filters, as shown in the figure below.

Note that operating Envoy without any listener filters is not untypical.



Envoy processes the listener filters before the network level filters. We can manipulate the connection metadata within the listener filters, usually to influence how later filters or clusters process the connection.

The listener filters operate on newly accepted sockets and can stop or subsequently continue execution to further filters. The order of the listener filters matters as Envoy processes them sequentially right after the listener accepts a socket and before the connection is created.

We can use the results from the listener filters to do the filter matching and select an appropriate network filter chain. For example, we could use an HTTP inspector listener filter to determine the HTTP protocol (HTTP/1.1 or HTTP/2). Based on the results, we can then select and run different network filter chains.

# 5.2 Filter chain matching

Filter chain matching allows us to specify the criteria for selecting a specific filter chain for a listener.

We can define multiple filter chains in the configuration and then select and execute them based on the destination port, server name, protocol, and other properties. For example, we could check which hostname is connecting and select a different filter chain. If hostname hello.com connects, we could choose a filter chain that presents the certificate for that specific hostname.

Before Envoy can start filter matching, it needs to have some data extracted from the received packet by listener filters. After that, for Envoy to select a specific filter chain, all match criteria must be fulfilled. For example, if we're matching on hostname and port, both values need to match for Envoy to select that filter chain.

The property matching order is as follows:

1. Destination port (when use\_original\_dst is used)
2. Destination IP address
3. Server name (SNI for TLS protocol)
4. Transport protocol
5. Application protocols (ALPN for TLS protocol)
6. Directly connected source IP address (this is only different from the source IP address if we're using a filter that overrides the source address, for example, the proxy protocol listener filter)
7. Source type (e.g., any, local, or external network)
8. Source IP address
9. Source port

Specific criteria, such as server name/SNI or IP addresses, also allow ranges or wildcards to be used. If using wildcard criteria in multiple filter chains, the most specific value will be matched.

For example, here's how the order from most specific to least specific match would look like for www.hello.com:

1. www.hello.com
2. \*.hello.com
3. \*.com
4. Any filter chain without the server name criteria

Here's an example of how we could configure filter chain matches using different properties:

filter\_chains:  
- filter\_chain\_match:  
 server\_names:  
 - "\*.hello.com"  
 filters:  
 ...  
- filter\_chain\_match:  
 source\_prefix\_ranges:  
 - address\_prefix: 192.0.0.1  
 prefix\_len: 32  
 filters:  
 ...  
- filter\_chain\_match:  
 transport\_protocol: tls  
 filters:  
 ...

Let's assume a TLS request comes in from the IP address 192.0.0.1 and has SNI set to v1.hello.com. Keeping the order in mind, the first filter chain match that satisfies all criteria is the server name match (v1.hello.com). Therefore the Envoy executes the filters under that match.

However, if the request comes in from the IP 192.0.0.1, it wouldn't be TLS, and the SNI doesn't match the \*.hello.com. Envoy will execute the second filter chain -- the one that matches the specific IP address.

# 5.3 HTTP inspector listener filter

The [HTTP inspector listener filter](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/http_inspector) (envoy.filters.listener.http\_inspector) allows us to detect if the application protocol appears to be HTTP. If the protocol is not HTTP, the listener filter will pass the packet.

If the application protocol is determined to be HTTP, it also detects the corresponding HTTP protocol (e.g., HTTP/1.x or HTTP/2).

We can check the result of the HTTP inspection filter using the application\_protocols field in the filter chain match.

Let's consider the following snippet:

...  
 listener\_filters:  
 - name: envoy.filters.listener.http\_inspector  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.listener.http\_inspector.v3.HttpInspector  
 filter\_chains:  
 - filter\_chain\_match:  
 application\_protocols: ["h2"]  
 filters:  
 - name: my\_http2\_filter  
 ...   
 - filter\_chain\_match:  
 application\_protocols: ["http/1.1"]  
 filters:  
 - name: my\_http1\_filter  
...

We've added the http\_inspector filter under the listener\_filters field to inspect the connection and determine the application protocol. If the HTTP protocol is HTTP/2 (h2c), Envoy matches the first network filter chain (starting with my\_http2\_filter).

Alternatively, if the downstream HTTP protocol is HTTP/1.1 (http/1.1), Envoy matches the second filter chain and runs the filter chain starting with the filter called my\_http1\_filter.

# 5.4 Original destination listener filter

The [Original destination](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/original_dst_filter) filter (envoy.filters.listener.original\_dst) reads the SO\_ORIGINAL\_DST socket option. This option is set when a connection has been redirected by an iptables REDIRECT or TPROXY target (if transparent option is set). The filter can be used in connection with a cluster with the ORIGINAL\_DST type.

When using the ORIGINAL\_DST cluster type, the requests get forwarded to upstream hosts as addressed by the redirection metadata without making any host discovery. Therefore defining any endpoints in the cluster doesn't make sense because the endpoint is taken from the original packet and isn't selected by a load balancer.

We can use Envoy as a generic proxy that forwards all requests to the original destination using this cluster type.

To use the ORIGINAL\_DST cluster, the traffic needs to reach Envoy through an iptables REDIRECT or TPROXY target.

...  
listener\_filters:  
- name: envoy.filters.listener.original\_dst  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.listener.original\_dst.v3.OriginalDst  
...  
clusters:  
 - name: original\_dst\_cluster  
 connect\_timeout: 5s  
 type: ORIGNAL\_DST  
 lb\_policy: CLUSTER\_PROVIDED

# 5.5 Original source listener filter

The [original source filter](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/original_src_filter)(envoy.filters.listener.original\_src) replicates the downstream (host connecting to Envoy) remote address of the connection on the upstream (host receiving requests from Envoy) side of Envoy.

For example, if we connect to Envoy with 10.0.0.1, Envoy connects to the upstream with source IP 10.0.0.1. The address is determined from the proxy protocol filter (explained next), or it can come from trusted HTTP headers.

- name: envoy.filters.listener.original\_src  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.listener.original\_src.v3.OriginalSrc  
 mark: 100

The filter also allows us to set the SO\_MARK option on the upstream connection's socket. The SO\_MARK option is used for marking each packet sent through the socket and allows us to do mark-based routing (we can match the mark later on).

The snippet above sets the mark to 100. Using this mark, we can ensure that non-local addresses may be routed back through the Envoy proxy when binding to the original source address.

# 5.6 Proxy protocol listener filter

The [Proxy protocol](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/proxy_protocol) listener filter (envoy.filters.listener.proxy\_protocol) adds support for the [HAProxy proxy protocol](https://www.haproxy.org/download/1.9/doc/proxy-protocol.txt).

Proxies use their IP stack to connect to remote servers and lose the source and destination information from the initial connection. The PROXY protocol allows us to chain proxies without losing client information. The protocol defines a way for communicating metadata about a connection over TCP before the main TCP stream. The metadata includes the source IP address.

Using this filter, Envoy can consume the metadata from the PROXY protocol and propagate it into an x-forwarded-for header, for example.

# 5.7 TLS inspector listener filter

The [TLS inspector](https://www.envoyproxy.io/docs/envoy/latest/configuration/listeners/listener_filters/tls_inspector) listener filter allows us to detect whether the transport appears to be TLS or plaintext. If the transport is TLS, it detects the server name indication (SNI) and/or the client's application-layer protocol negotiation (ALPN).

**What is SNI?**

SNI or server name indication is an extension to the TLS protocol, and it tells us which hostname is connecting at the start of the TLS handshake process. We can serve multiple HTTPS services (with different certificates) on the same IP address and port using SNI. If a client connects with hostname "hello.com", the server can present the certificate for that hostname. Similarly, if the client connects with "example.com" the server offers that certificate.

**What is ALPN?**

ALPN or application-layer protocol negotiation is an extension to the TLS protocol that allows the application layer to negotiate which protocol should be performed over a secure connection without making additional round trips. Using ALPN we can determine whether the client is speaking HTTP/1.1 or HTTP/2.

We can use SNI and ALPN values to match filter chains using the server\_names (for SNI) and/or application\_protocols (for ALPN) fields.

The snippet below shows how we could use the application\_protocols and server\_names to execute different filter chains.

...  
 listener\_filters:  
 - name: "envoy.filters.listener.tls\_inspector"  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.listener.tls\_inspector.v3.TlsInspector  
 filter\_chains:  
 - filter\_chain\_match:  
 application\_protocols: ["h2c"]  
 filters:  
 - name: some\_filter  
 ...   
 - filter\_chain\_match:  
 server\_names: "something.hello.com"  
 transport\_socket:  
 ...  
 filters:  
 - name: another\_filter  
...

# 6.0 Logging

In the logging module, we’ll learn about different types of and approaches to logging in Envoy.

Whether it’s used as a traffic gateway or a side car in the service mesh, Envoy is in a unique position to reveal what’s going on in your network. A common path to understanding is logging, be it for analytics, auditing, or troubleshooting. Logging also carries a problem of volume and potential to reveal secrets.

By the end of this module, you’ll learn what logging options exist in Envoy, including how logs can be structured and filtered. as well as where they can be written to.

# 6.1 Access logging

## What is access logging?

Whenever you open your browser and visit Google or other websites, the server on the other side collects the information about your visit. Specifically, it’s collecting and storing the data about web pages you requested from the server. In most cases, this data includes the origin (i.e., host information), date and time you requested the page, the request properties (method, path, headers, body, etc.), the status the server returns, size of the request, and more. All this data typically gets stored in text files called **access logs**.

Typically, access log entries from web servers or proxies follow a standardized common logging format. Different proxies and servers can use their own default access log formats. Envoy has its default logging format. We can customize the default format and configure it to write out the logs in the same format as other servers such as Apache or NGINX. Having the same access log format allows us to use different servers together and combine data logging and analysis using a single tool.

This module will explain how access logging works in Envoy and how to configure and customize it.

## Capturing and reading access logs

We can configure capture of any access requests made to the Envoy proxy and write them to so-called access logs. Let’s look at an example of a couple of access log entries:

[2021-11-01T20:37:45.204Z] "GET / HTTP/1.1" 200 - 0 3 0 - "-" "curl/7.64.0" "9c08a41b-805f-42c0-bb17-40ec50a3377a" "localhost:10000" "-"  
[2021-11-01T21:08:18.274Z] "POST /hello HTTP/1.1" 200 - 0 3 0 - "-" "curl/7.64.0" "6a593d31-c9ac-453a-80e9-ab805d07ae20" "localhost:10000" "-"  
[2021-11-01T21:09:42.717Z] "GET /test HTTP/1.1" 404 NR 0 0 0 - "-" "curl/7.64.0" "1acc3559-50eb-463c-ae21-686fe34abbe8" "localhost:10000" "-"

The output contains three different log entries and follows the same default log format. The default log format looks like this:

[%START\_TIME%] "%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%"  
%RESPONSE\_CODE% %RESPONSE\_FLAGS% %BYTES\_RECEIVED% %BYTES\_SENT% %DURATION%  
%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% "%REQ(X-FORWARDED-FOR)%" "%REQ(USER-AGENT)%"  
"%REQ(X-REQUEST-ID)%" "%REQ(:AUTHORITY)%" "%UPSTREAM\_HOST%"

The values such as %RESPONSE\_FLAGS%, %REQ(:METHOD)% and others are called **command operators**.

### Command operators

The command operators extract the relevant data and insert it in the log entry for both TCP and HTTP. If the values are not set or not available (for example, RESPONSE\_CODE in TCP) the logs will contain the character - (or "-" for JSON logs).

Each command operator starts and ends with the character %, for example, %START\_TIME%. If the command operator accepts any parameters, then we can provide them in the parentheses. For example, if we wanted to log only day, month, and year using the START\_TIME command operator, then we could configure it by specifying the values in the parentheses like this: %START\_TIME(%d-%m-%Y)%.

Let's look at the different command operators. We've tried to group them into separate tables based on their common properties.

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| START\_TIME | Request start time including milliseconds. | %START\_TIME(%Y/%m/%dT%H:%M:%S%z ^%)% |
| PROTOCOL | Protocol (either HTTP/1.1, HTTP/2 or HTTP/3). | %PROTOCOL% |
| RESPONSE\_CODE | HTTP response code. Response code gets set to 0 if the downstream client disconnected. | %RESPONSE\_CODE% |
| RESPONSE\_CODE\_DETAILS | Additional information about the HTTP response (e.g. who set it and why). | %RESPONSE\_CODE\_DETAILS% |
| CONNECTION\_TERMINATION\_DETAILS | Provides additional information about why Envoy terminated the connection for L4 reasons. | %CONNECTION\_TERMINATION\_DETAILS% |
| ROUTE\_NAME | Name of the route. | %ROUTE\_NAME% |
| CONNECTION\_ID | An identifier for the downstream connection. It can be used to cross-reference TCP access logs across multiple log sinks or cross-reference timer-based reports for the same connection. The identifier is unique with a high likelihood within an execution but can duplicate across multiple instances or between restarts. | %CONNECTION\_ID% |
| GRPC\_STATUS | gRPC status code including text message and a number. | %GRPC\_STATUS% |
| HOSTNAME | The system hostname. | %HOSTNAME% |
| LOCAL\_REPLY\_BODY | The body text for the requests rejected by Envoy. | %LOCAL\_REPLY\_BODY% |
| FILTER\_CHAIN\_NAME | The network filter chain name of the downstream connection. | %FILTER\_CHAIN\_NAME% |

#### Sizes

This group contains all command operators representing sizes — from request and response header bytes to bytes received and sent.

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| REQUEST\_HEADER\_BYTES | Uncompressed bytes of request headers. | %REQUEST\_HEADER\_BYTES% |
| RESPONSE\_HEADERS\_BYTES | Uncompressed bytes of response headers. | %RESPONSE\_HEADERS\_BYTES% |
| RESPONSE\_TRAILERS\_BYTES | Uncompressed bytes of response trailers. | %RESPONSE\_TRAILERS\_BYTES% |
| BYTES\_SENT | Body bytes sent for HTTP and downstream bytes sent on connection for TCP. | %BYTES\_SENT% |
| BYTES\_RECEIVED | Body bytes received. | %BYTES\_RECEIVED% |
| UPSTREAM\_WIRE\_BYTES\_SENT | Total number of bytes sent upstream by the HTTP stream. | %UPSTREAM\_WIRE\_BYTES\_SENT% |
| UPSTREAM\_WIRE\_BYTES\_RECEIVED | Total number of bytes received from the upstream HTTP stream, | %UPSTREAM\_WIRE\_BYTES\_RECEIVED% |
| UPSTREAM\_HEADER\_BYTES\_SENT | Number of header bytes sent upstream by the HTTP stream. | %UPSTREAM\_HEADER\_BYTES\_SENT% |
| UPSTREAM\_HEADER\_BYTES\_RECEIVED | Number of header bytes received from the upstream by the HTTP stream. | %UPSTREAM\_HEADER\_BYTES\_RECEIVED% |
| DOWNSTREAM\_WIRE\_BYTES\_SENT | Total number of bytes sent downstream by the HTTP stream. | %DOWNSTREAM\_WIRE\_BYTES\_SENT% |
| DOWNSTREAM\_WIRE\_BYTES\_RECEIVED | Total number of bytes received from the downstream by the HTTP stream. | %DOWNSTREAM\_WIRE\_BYTES\_RECEIVED% |
| DOWNSTREAM\_HEADER\_BYTES\_SENT | Number of header bytes sent downstream by the HTTP stream. | %DOWNSTREAM\_HEADER\_BYTES\_SENT% |
| DOWNSTREAM\_HEADER\_BYTES\_RECEIVED | Number of header bytes received from the downstream by the HTTP stream. | %DOWNSTREAM\_HEADER\_BYTES\_RECEIVED% |

#### Durations

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| DURATION | Total duration of the request (in milliseconds) from the start time to the last byte out. | %DURATION% |
| REQUEST\_DURATION | Total duration of the request (in milliseconds) from the start time to the last byte of the request received from downstream. | %REQUEST\_DURATION% |
| REQUEST\_TX\_DURATION | Total duration of the request (in milliseconds) from the start time to the last byte sent upstream. | %REQUEST\_TX\_DURATION% |
| RESPONSE\_DURATION | Total duration of the request (in milliseconds) from the start time to the first byte read from the upstream host. | %RESPONSE\_DURATION% |
| RESPONSE\_TX\_DURATION | Total duration of the request (in milliseconds) from the first byte read from the upstream host to the last byte sent downstream. | %RESPONSE\_TX\_DURATION% |

#### Response flags

The RESPONSE\_FLAGS command operator contains additional details about the response or connection. The following list shows the response flags' values and their meaning for HTTP and TCP connections.

**HTTP and TCP**

* UH: No healthy upstream hosts in an upstream cluster in addition to 503 response code.
* UF: Upstream connection failure in addition to 503 response code.
* UO: Upstream overflow (circuit breaking) in addition to 503 response code.
* NR: No route configured for a given request in addition to 404 response code or no matching filter chain for a downstream connection.
* URX: The request was rejected because the upstream retry limit (HTTP) or maximum connection attempts (TCP) was reached.
* NC: Upstream cluster not found.
* DT: When a request or connection exceeded max\_connection\_duration or max\_downstream\_connection\_duration.

**HTTP only**

* DC: Downstream connection termination.
* LH: Local service failed health check request in addition to 503 response code.
* UT: Upstream request timeout in addition to 504 response code.
* LR: Connection local reset in addition to 503 response code.
* UR: Upstream remote reset in addition to 503 response code.
* UC: Upstream connection termination in addition to 503 response code.
* DI: The request processing was delayed for a period specified via fault injection.
* FI: The request was aborted with a response code specified via fault injection.
* RL: The request was rate-limited locally by the HTTP rate limit filter in addition to the 429 response code.
* UAEX: The request was denied by the external authorization service.
* RLSE: The request was rejected because there was an error in the rate limit service.
* IH: The request was rejected because it set an invalid value for a strictly checked header in addition to 400 response code.
* SI: Stream idle timeout in addition to 408 response code.
* DPE: The downstream request had an HTTP protocol error.
* UPE: The upstream response had an HTTP protocol error.
* UMSDR: The upstream request reached max stream duration.
* OM: Overload Manager terminated the request.

#### Upstream information

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| UPSTREAM\_HOST | Upstream host URL or tcp://ip:port for TCP connections. | %UPSTREAM\_HOST% |
| UPSTREAM\_CLUSTER | Upstream cluster to which the upstream host belongs. If runtime feature envoy.reloadable\_features.use\_observable\_cluster\_name is enabled, then alt\_stat\_name will be used if provided. | %UPSTREAM\_CLUSTER% |
| UPSTREAM\_LOCAL\_ADDRESS | Local address of the upstream connection. If it's an IP address, then it includes both address and port. | %UPSTREAM\_LOCAL\_ADDRESS% |
| UPSTREAM\_TRANSPORT\_FAILURE\_REASON | Provides the failure reason from the transport socket if connection failed due to transport socket. | %UPSTREAM\_TRANSPORT\_FAILURE\_REASON% |

#### Downstream information

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| DOWNSTREAM\_REMOTE\_ADDRESS | Remote address of the downstream connection. If it's an IP address, then it includes both address and port. | %DOWNSTREAM\_REMOTE\_ADDRESS% |
| DOWNSTREAM\_REMOTE\_ADDRESS\_WITHOUT\_PORT | Remote address of the downstream connection. If it's an IP address, then it includes the address only. | %DOWNSTREAM\_REMOTE\_ADDRESS\_WITHOUT\_PORT% |
| DOWNSTREAM\_DIRECT\_REMOTE\_ADDRESS | Direct remote address of the downstream connection. If it's an IP address, then it includes both address and port. | %DOWNSTREAM\_DIRECT\_REMOTE\_ADDRESS% |
| DOWNSTREAM\_DIRECT\_REMOTE\_ADDRESS\_WITHOUT\_PORT | Direct remote address of the downstream connection. If it's an IP address, then it includes address only. | %DOWNSTREAM\_DIRECT\_REMOTE\_ADDRESS\_WITHOUT\_PORT% |
| DOWNSTREAM\_LOCAL\_ADDRESS | Local address of the downstream connection. If it's an IP address, then it includes both address and port. If the original connection was redirected by iptables REDIRECT, then this value represents the original destination address restored by the original destination filter. If redirected by iptables TPROXY and the listener's transparent option was set to true, then this represents the original destination address and port. | %DOWNSTREAM\_LOCAL\_ADDRESS% |
| DOWNSTREAM\_LOCAL\_ADDRESS\_WITHOUT\_PORT | Same as DOWNSTREAM\_LOCAL\_ADDRESS excluding port if the address is an IP address. | %DOWNSTREAM\_LOCAL\_ADDRESS\_WITHOUT\_PORT% |
| DOWNSTREAM\_LOCAL\_PORT | Similar to DOWNSTREAM\_LOCAL\_ADDRESS\_WITHOUT\_PORT but only extracts the port portion of the DOWNSTREAM\_LOCAL\_ADDRESS. | %DOWNSTREAM\_LOCAL\_PORT% |

#### Headers and trailers

The REQ, RESP, and TRAILER command operator allows us to extract request, response, and trailer header information and include it in the logs.

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| REQ(X?Y):Z | HTTP request header where X is the main HTTP header, Y is the alternative one, and Z is an optional parameter denoting string truncation up to Z characters long. If the value from header X is not set, then request header Y is used. If none of the headers are present, - will be in the logs. | %REQ(HELLO?BYE):5% includes the value of header hello. If not set, uses the value from header bye. It truncates the value to 5 characters. |
| RESP(X?Y):Z | Same as REQ but taken from HTTP response headers. | %RESP(HELLO?BYE):5% includes the value of header hello. If not set, uses the value from header bye. It truncates the value to 5 characters. |
| TRAILER(X?Y):Z | Same as REQ but taken from HTTP response trailers | %TRAILER(HELLO?BYE):5% includes the value of header hello. If not set, uses the value from header bye. It truncates the value to 5 characters. |

#### Metadata

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| DYNAMIC\_METADATA(NAMESPACE:KEY\*):Z | Dynamic metadata info, where NAMESPACE is the filter used when setting the metadata. KEY is an optional lookup key in the namespace with the option of specifying nested keys separated by :. Z is an optional parameter denoting string truncation up to Z characters long. | For example, the my\_filter: {"my\_key": "hello", "json\_object": {"some\_key": "foo"}} metadata can be logged using %DYNAMIC\_METADATA(my\_filter)%. To log a specific key, we could write %DYNAMIC\_METADATA(my\_filter:my\_key)%. |
| CLUSTER\_METADATA(NAMESPACE:KEY\*):Z | Upstream cluster metadata info, where NAMESPACE is the filter namespace used when setting the metadata, KEY is an optional lookup key in the namespace with the option of specifying nested keys separated by :. Z is an optional parameter denoting string truncation up to Z characters long. | See example for DYNAMIC\_METADATA |
| FILTER\_STATE(KEY:F):Z | Filter state info, where the KEY is required to look up the filter state object. The serialized proto will be logged as a JSON string if possible. If the serialized proto is unknown, then it will be logged as a protobuf debug string. F is an optional parameter indicating which method FilterState uses for serialization. If PLAIN is set, then the filter state object will be serialized as an unstructured string. If TYPED is set or no F is provided, then the filter state object will be serialized as a JSON string. Z is an optional parameter denoting string truncation up to Z characters long. | %FILTER\_STATE(my\_key:PLAIN):10% |

#### TLS

|  |  |  |
| --- | --- | --- |
| Command operator | Description | Example |
| REQUESTED\_SERVER\_NAME | String value set on SSL connection socket for Server Name Indication (SNI). | %REQUESTED\_SERVER\_NAME% |
| DOWNSTREAM\_LOCAL\_URI\_SAN | The URIs present in the SAN of the local certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_LOCAL\_URI\_SAN% |
| DOWNSTREAM\_PEER\_URI\_SAN | The URIs present in the SAN of the peer certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_URI\_SAN% |
| DOWNSTREAM\_LOCAL\_SUBJECT | The subject present in the local certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_LOCAL\_SUBJECT% |
| DOWNSTREAM\_PEER\_SUBJECT | The subject present in the peer certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_SUBJECT% |
| DOWNSTREAM\_PEER\_ISSUER | The issuer present in the peer certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_ISSUER% |
| DOWNSTREAM\_TLS\_SESSION\_ID | The session ID for the established downstream TLS connection. | %DOWNSTREAM\_TLS\_SESSION\_ID% |
| DOWNSTREAM\_TLS\_CIPHER | The OpenSSL name for the set of ciphers used to establish the downstream TLS connection. | %DOWNSTREAM\_TLS\_CIPHER% |
| DOWNSTREAM\_TLS\_VERSION | The TLS version (TLSv1.2 or TLSv1.3) used to establish the downstream TLS connection. | %DOWNSTREAM\_TLS\_VERSION% |
| DOWNSTREAM\_PEER\_FINGERPRINT\_256 | The hex-encoded SHA256 fingerprint of the client certificated used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_FINGERPRINT\_256% |
| DOWNSTREAM\_PEER\_FINGERPRINT\_1 | The hex-encoded SHA1 fingerprint of the client certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_FINGERPRINT\_1% |
| DOWNSTREAM\_PEER\_SERIAL | The serial number of the client certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_SERIAL% |
| DOWNSTREAM\_PEER\_CERT | The client certificate in the URL-safe encoded PEM format used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_CERT% |
| DOWNSTREAM\_PEER\_CERT\_V\_START | The validity start date of the client certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_CERT\_V\_START%. Can be customized like START\_TIME |
| DOWNSTREAM\_PEER\_CERT\_V\_END | The validity end date of the client certificate used to establish the downstream TLS connection. | %DOWNSTREAM\_PEER\_CERT\_V\_END%. Can be customized like START\_TIME |

# 6.2 Configuring access loggers

We can configure access loggers on the HTTP or TCP filter level and the listener level. We can also configure multiple access logs with different logging formats and logging sinks. A **logging sink** is an abstract term for the location the logs write to, for example to the console (stdout, stderr), a file, or a network service.

A scenario in which we’d configure multiple access logs is when we’d like to see high-level information in the console (standard out) and full request details written to a file on the disk. The field used to configure the access loggers is called access\_log.

Let's look at an example of enabling access logging to standard out (StdoutAccessLog) on the HTTP connection manager (HCM) level:

- filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog

Envoy aims to have portable and extensible configuration: typed config. A side effect of this is verbose names for configuration. For example, to enable access logging, we find the name of the HTTP configuration type and then the type corresponding to the console (StdoutAccessLog).

The StdoutAccessLog configuration writes the log entries to the standard out (the console). Other supported access logging sinks are the following:

* File (FileAccessLog)
* gRPC (HttpGrpcAccessLogConfig and TcpGrpcAccessLogConfig)
* Standard error (StderrAccessLog)
* Wasm (WasmAccessLog)
* Open Telemetry

The file access log allows us to write log entries to a file we specify in the configuration. For example:

- filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.file  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.file.v3.FileAccessLog  
 path: ./envoy-access-logs.log

Note the change in the name (envoy.access\_loggers.file) and the type (file.v3.FileAccessLog). Additionally, we've provided the path where we want Envoy to store the access logs.

The gRPC access logging sink sends the logs to an HTTP or TCP gRPC logging service. To use the gRPC logging sink, we have to build a gRPC server with an endpoint that implements the [MetricsService](https://github.com/envoyproxy/envoy/blob/main/api/envoy/service/metrics/v3/metrics_service.proto), specifically the StreamMetrics function. Then, Envoy can connect to the gRPC server and send the logs to it.

Earlier, we mentioned the default access log format that's comprised of different command operators:

[%START\_TIME%] "%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%"  
%RESPONSE\_CODE% %RESPONSE\_FLAGS% %BYTES\_RECEIVED% %BYTES\_SENT% %DURATION%  
%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% "%REQ(X-FORWARDED-FOR)%" "%REQ(USER-AGENT)%"  
"%REQ(X-REQUEST-ID)%" "%REQ(:AUTHORITY)%" "%UPSTREAM\_HOST%"

The format of the log entries is configurable and can be modified using the log\_format field. Using the log\_format, we can configure which values the log entry includes and specify whether we want logs in plain text or JSON format.

Let's say we want to log only the start time, response code, and the user agent. We'd configure it like this:

- filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 log\_format:  
 text\_format\_source:  
 inline\_string: "%START\_TIME% %RESPONSE\_CODE% %REQ(USER-AGENT)%"

A sample log entry using the above format would look like this:

2021-11-01T21:32:27.170Z 404 curl/7.64.0

Similarly, instead of providing a text format, we can also set up the JSON format string if we want the logs to be in a structured format such as JSON.

To use the JSON format, we have to provide a format dictionary instead of a single string, as in plain text format.

Here's an example of using the same log format but writing the log entries in JSON instead:

- filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 log\_format:  
 json\_format:  
 start\_time: "%START\_TIME%"  
 response\_code: "%RESPONSE\_CODE%"  
 user\_agent: "%REQ(USER-AGENT)%"

The above snippet would generate the following log entry:

{"user\_agent":"curl/7.64.0","response\_code":404,"start\_time":"2021-11-01T21:37:59.979Z"}

Certain command operators, such as FILTER\_STATE or DYNAMIC\_METADATA, might produce nested JSON log entries.

The log format can also use formatter plugins specified through the formatters field. There are two known formatter extensions in the current version: the metadata (envoy.formatter.metadata) and request without query (envoy.formatter.req\_without\_query) extension.

The metadata formatter extension implements the METADATA command operator that allows us to output different types of metadata (DYNAMIC, CLUSTER, or ROUTE).

Similarly, the req\_without\_query formatter allows us to use the REQ\_WITHOUT\_QUERY command operator which works the same way as the REQ command operator but removes the query string. The command operator is used to avoid logging any sensitive information into the access log.

Here's an example of how to provide a formatter and how to use it in the inline\_string:

- filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.network.http\_connection\_manager.v3.HttpConnectionManager  
 stat\_prefix: ingress\_http  
 access\_log:  
 - name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 log\_format:  
 text\_format\_source:  
 inline\_string: "[%START\_TIME%] %REQ(:METHOD)% %REQ\_WITHOUT\_QUERY(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%"  
 formatters:  
 - name: envoy.formatter.req\_without\_query  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.formatter.req\_without\_query.v3.ReqWithoutQuery

The above configuration with this request curl localhost:10000/?hello=1234 would generate a log entry that doesn't include the query parameters (hello=1234):

[2021-11-01T21:48:55.941Z] GET / HTTP/1.1

# 6.3 Access log filtering

Another feature of access logging in Envoy is the ability to specify filters that determine whether the access log needs to be written or not. For example, we could have an access log filter that logs only the 500 status codes, only log requests that took more than 5 seconds, and so on. The table below shows supported access log filters.

|  |  |
| --- | --- |
| Access log filter name | Description |
| status\_code\_filter | Filters on status code value. |
| duration\_filter | Filters on total request duration in milliseconds. |
| not\_health\_check\_filter | Filters for requests that are not health check requests. |
| traceable\_filter | Filters for requests that are traceable. |
| runtime\_filter | Filters for random sampling of requests. |
| and\_filter | Performs a logical "and" operation on the result of each filter in the list of filters. Filters are evaluated sequentially. |
| or\_filter | Performs a logical "or" operation on the result of each filter in the list of filters. Filters are evaluated sequentially. |
| header\_filter | Filters requests based on the presence or value of a request header. |
| response\_flag\_filter | Filters requests that received responses with an Envoy response flag set. |
| grpc\_status\_filter | Filters gRPC requests based on their response status. |
| extension\_filter | Use an extension filter that's statically registered at runtime. |
| metadata\_filter | Filters based on matching dynamic metadata. |

Each filter has different properties that we have an option to set. Here's a snippet that shows how to use the status code, header, and an *and* filter:

...  
access\_log:  
- name: envoy.access\_loggers.stdout  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.access\_loggers.stream.v3.StdoutAccessLog  
 filter:  
 and\_filter:  
 filters:  
 header\_filter:  
 header:  
 name: ":method"  
 string\_match:  
 exact: "GET"  
 status\_code\_filter:  
 comparison:  
 op: GE  
 value:  
 default\_value: 400  
...

The above snippet writes a log entry to the standard out for all GET requests with response codes greater than or equal 400.

# 6.4 Envoy component logs

So far, we’ve talked about logs generated as a result of sending requests to Envoy. However, Envoy also produces logs as part of startup and during execution.

We can see the Envoy component logs each time we run Envoy:

...  
[2021-11-03 17:22:43.361][1678][info][main] [source/server/server.cc:368] initializing epoch 0 (base id=0, hot restart version=11.104)  
[2021-11-03 17:22:43.361][1678][info][main] [source/server/server.cc:370] statically linked extensions:  
[2021-11-03 17:22:43.361][1678][info][main] [source/server/server.cc:372] envoy.filters.network: envoy.client\_ssl\_auth, envoy.echo, envoy.ext\_authz, envoy.filters.network.client\_ssl\_auth  
...

The default format string for a component log is [%Y-%m-%d %T.%e][%t][%l][%n] [%g:%#] %v. The first part of the format string represents the date and time, followed by thread ID (%t), the log level of the message (%l), logger name (%n), the relative path of the source file and line number (%g:%#), and the actual log message (%v).

When starting Envoy, we can use the --log-format command-line option to customize the format. For example, if we wanted to log the time logger name, source function name, and the log message, then we could write the format string like so: [%T.%e][%n][%!] %v.

Then, when starting Envoy, we can set the format string as follows:

func-e run -c someconfig.yaml --log-format '[%T.%e][%n][%!] %v'

If we use the format string, the log entries look like this:

[17:43:15.963][main][initialize] response trailer map: 160 bytes: grpc-message,grpc-status  
[17:43:15.965][main][createRuntime] runtime: {}  
[17:43:15.965][main][initialize] No admin address given, so no admin HTTP server started.  
[17:43:15.966][config][initializeTracers] loading tracing configuration  
[17:43:15.966][config][initialize] loading 0 static secret(s)  
[17:43:15.966][config][initialize] loading 0 cluster(s)  
[17:43:15.966][config][initialize] loading 1 listener(s)  
[17:43:15.969][config][initializeStatsConfig] loading stats configuration  
[17:43:15.969][runtime][onRtdsReady] RTDS has finished initialization  
[17:43:15.969][upstream][maybeFinishInitialize] cm init: all clusters initialized  
[17:43:15.969][main][onRuntimeReady] there is no configured limit to the number of allowed active connections. Set a limit via the runtime key overload.global\_downstream\_max\_connections  
[17:43:15.970][main][operator()] all clusters initialized. initializing init manager  
[17:43:15.970][config][startWorkers] all dependencies initialized. starting workers  
[17:43:15.971][main][run] starting main dispatch loop

Envoy features multiple loggers, and for each logger (e.g. main, config, http, ...), we can control the logging level (info, debug, trace). We can look at the names of all active loggers if we enable the Envoy admin interface and send a request to /logging path. Another way to look at all available loggers is via the [source code](https://github.com/envoyproxy/envoy/blob/82261f5a401418df13626ca3fa52fa65fea10c81//source/common/common/logger.h).

Here's how the default output from /logging endpoint looks:

active loggers:  
 admin: info  
 alternate\_protocols\_cache: info  
 aws: info  
 assert: info  
 backtrace: info  
 cache\_filter: info  
 client: info  
 config: info  
 connection: info  
 conn\_handler: info  
 decompression: info  
 dns: info  
 dubbo: info  
 envoy\_bug: info  
 ext\_authz: info  
 rocketmq: info  
 file: info  
 filter: info  
 forward\_proxy: info  
 grpc: info  
 hc: info  
 health\_checker: info  
 http: info  
 http2: info  
 hystrix: info  
 init: info  
 io: info  
 jwt: info  
 kafka: info  
 key\_value\_store: info  
 lua: info  
 main: info  
 matcher: info  
 misc: info  
 mongo: info  
 quic: info  
 quic\_stream: info  
 pool: info  
 rbac: info  
 redis: info  
 router: info  
 runtime: info  
 stats: info  
 secret: info  
 tap: info  
 testing: info  
 thrift: info  
 tracing: info  
 upstream: info  
 udp: info  
 wasm: info

Notice that the default logging level for every logger is set to info. The other log levels are the following:

* trace
* debug
* info
* warning/warn
* error
* critical
* off

To configure log levels, we can use the --log-level option or the --component-log-level to control the log level for each component separately. The component log levels can be written in the format of log\_name:log\_level. If we're setting log levels for multiple components, then we separate them with a comma. For example: upstream:critical,secret:error,router:trace.

For example, to set the main log level to trace, config log level to error, and turn off all other loggers, we could type the following:

func-e run -c someconfig.yaml --log-level off --component-log-level main:trace,config:error

By default, all Envoy application logs are written to the standard error (stderr). To change that, we can provide an output file using the --log-path option:

func-e run -c someconfig.yaml --log-path app-logs.log

In one of the labs, we'll also show how Envoy can be configured to write application logs to the Google Cloud operations suite (formerly known as Stackdriver).

# 7.0 Admin interface

In the admin interface module, we’ll learn about the administrative interfaces that Envoy exposes and the different endpoints we can use to retrieve configuration and stats, as well as perform other administrative tasks.

By the end of this module, you’ll understand how to enable the admin interface and the different tasks we can perform through it.

# 7.1 Enabling admin interface

Throughout the course, we've referred to the administration interface a couple of times. Envoy exposes an admin interface that allows us to modify Envoy and get a view and query for the metrics and configuration.

The admin interface consists of a REST API with multiple endpoints and a simple UI, as shown in the figure below.



The admin interface must be explicitly enabled using the admin field. For example:

admin:  
 address:  
 socket\_address:  
 address: 127.0.0.1  
 port\_value: 9901

Be careful when enabling the administrative interface. Anyone with access to the admin interface can perform destructive operations, such as shutting down the server (the /quitquitquit endpoint). We're also potentially giving them access to private information (metrics, cluster names, certificate information, and so on). Currently (Envoy version 1.20), the admin endpoint is unsecured, and there's no way to configure authentication or TLS. There's a work item in progress that will restrict access only to trusted IPs and client certificates to ensure transport security.

Until that work is completed, access to the admin interface should be allowed only via a secure network and only from hosts attached to that secure network. We have an option of allowing access to the admin interface via localhost only, as shown in the above configuration. Alternatively, if you decide to allow access from remote hosts, then make sure you also set up firewall rules.

In the upcoming lessons, we'll look at the different features of the admin interface in more detail.

# 7.2 Configuration dump

The /config\_dump endpoint is a quick way to show the currently loaded Envoy configuration as JSON-serialized proto messages.

Envoy outputs the configuration for the following components, and in the order presented below:

* bootstrap
* clusters
* endpoints
* listeners
* scoped routes
* routes
* secrets

### Including the EDS config

To output the endpoint discovery service (EDS) configuration, we can add the ?include\_eds parameter to the query.

### Filtering the output

Similarly, we can filter the output by providing the resources we want to include and a mask to return a subset of fields.

For example, to output only a static cluster configuration, we can use the static\_clusters field from [ClustersConfigDump proto](https://www.envoyproxy.io/docs/envoy/latest/api-v3/admin/v3/config_dump.proto#envoy-v3-api-msg-admin-v3-clustersconfigdump) in the resource query parameter:

$ curl localhost:9901/config\_dump?resource=static\_clusters  
{  
 "configs": [  
 {  
 "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster",  
 "cluster": {  
 "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster",  
 "name": "instance\_1",  
 },  
 ...  
 {  
 "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster",  
 "cluster": {  
 "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster",  
 "name": "instance\_2",  
...

#### Using the mask parameter

To narrow the output further, we can specify the field in the mask parameter. For example, to show only the connect\_timeout values for every cluster:

$ curl localhost:9901/config\_dump?resource=static\_clusters&mask=cluster.connect\_timeout  
{  
 "configs": [  
 {  
 "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster",  
 "cluster": {  
 "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster",  
 "connect\_timeout": "5s"  
 }  
 },  
 {  
 "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster",  
 "cluster": {  
 "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster",  
 "connect\_timeout": "5s"  
 }  
 },  
 {  
 "@type": "type.googleapis.com/envoy.admin.v3.ClustersConfigDump.StaticCluster",  
 "cluster": {  
 "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster",  
 "connect\_timeout": "1s"  
 }  
 }  
 ]  
}

#### Using regular expressions

Another filtering option is to specify a regular expression that matches the names of loaded configurations. For example, to output all listeners whose name field matches the regular expression .\*listener.\*, we could write the following:

$ curl localhost:9901/config\_dump?resource=static\_clusters&name\_regex=.\*listener.\*  
  
{  
 "configs": [  
 {  
 "@type": "type.googleapis.com/envoy.admin.v3.ListenersConfigDump.StaticListener",  
 "listener": {  
 "@type": "type.googleapis.com/envoy.config.listener.v3.Listener",  
 "name": "listener\_0",  
 "address": {  
 "socket\_address": {  
 "address": "0.0.0.0",  
 "port\_value": 10000  
 }  
 },  
 "filter\_chains": [  
 {}  
 ]  
 },  
 "last\_updated": "2021-11-15T20:06:51.208Z"  
 }  
 ]  
}

Similarly, the /init\_dump endpoint lists current information of unready targets of various Envoy components. Like the configuration dump, we can use the mask query parameter to filter for particular fields.

## Certificates

The /certs outputs all loaded TLS certificates. The data includes the certificate file name, serial number, subject alternate names, and days until expiration. The result is in JSON format, and it follows the [admin.v3.Certificates proto](https://www.envoyproxy.io/docs/envoy/latest/api-v3/admin/v3/certs.proto#envoy-v3-api-msg-admin-v3-certificates).

# 7.3 Statistics

The primary endpoint for the admin interface’s statistics output is accessed through the /stats endpoint. This input is typically used for debugging. We can access the endpoint by either sending a request to /stats endpoint or accessing the same path from the administrative UI.

The endpoint supports filtering the returned stats using the filter query parameter and a regular expression.

Another dimension to filter the output is using the usedonly query parameter. When used, it will output only the statistics that Envoy has updated. For example, counters that have been incremented at least once, gauges changed at least once, and histograms added to at least once.

By default, the stats are written in the StatsD format. Each stat is written to a separate line and the stat name (e.g., cluster\_manager.active\_clusters) is followed by the stat value (e.g., 15).

For example:

...  
cluster\_manager.active\_clusters: 15  
cluster\_manager.cluster\_added: 3  
cluster\_manager.cluster\_modified: 4  
...

The format query parameter controls the output format. Setting it to json will output the stats in JSON format. This format is typically used if we want to access and parse the stats programmatically.

The second format is the Prometheus format (e.g., format=prometheus). This option formats the status in the Prometheus format and can be used to integrate with a Prometheus server. Alternatively, we can use the /stats/prometheus endpoint to get the same output.

## Memory

The /memory endpoint will output the current memory allocation and heap usage in bytes. It's a subset of information /stats endpoint prints out.

$ curl localhost:9901/memory  
{  
 "allocated": "5845672",  
 "heap\_size": "10485760",  
 "pageheap\_unmapped": "0",  
 "pageheap\_free": "3186688",  
 "total\_thread\_cache": "80064",  
 "total\_physical\_bytes": "12699350"  
}

## Reset counters

Sending a POST request to /reset\_counters resets all counters to zero. Note that this won't reset or drop any data sent to statsd. It affects only the output of the /stats endpoint. The /stats endpoint and the /reset\_counters endpoint can be used during debugging.

## Server information and status

The /server\_info endpoint outputs the information about the running Envoy server. This includes the version, state, configuration path, log level information, uptime, node information, and more.

The [admin.v3.ServerInfo](https://www.envoyproxy.io/docs/envoy/latest/api-v3/admin/v3/server_info.proto#envoy-v3-api-msg-admin-v3-serverinfo) proto explains the different fields returned by the endpoint.

The /ready endpoint returns a string and an error code reflecting the state of the Envoy. If Envoy is live and ready to accept connections, then it returns the HTTP 200 and the string LIVE. Otherwise, the output will be an HTTP 503. This endpoint can be used as a readiness check.

The /runtime endpoint outputs all runtime values in a JSON format. The output includes the list of active runtime override layers and each key's stack of layer values. The values can also be modified by sending a POST request to the /runtime\_modify endpoint and specifying key/value pairs — for example, POST /runtime\_modify?my\_key\_1=somevalue.

The /hot\_restart\_version endpoint, together with the --hot-restart-version flag, can be used to determine whether the new binary and the running binary are hot restart compatible.

**Hot restart** is Envoy's ability to "hot" or "live" restart itself. This means that Envoy can fully reload itself (and configuration) without dropping any existing connections.

## Hystrix event stream

The /hystrix\_event\_stream endpoint is meant to be used as the stream source for the [Hystrix dashboard](https://github.com/Netflix-Skunkworks/hystrix-dashboard/wiki). Sending a request to the endpoint will trigger a stream of statistics from Envoy in the format that's expected by the Hystrix dashboard.

Note that we have to configure the Hystrix stats sync in the bootstrap configuration for the endpoint to work.

For example:

stats\_sinks:   
 - name: envoy.stat\_sinks.hystrix  
 typed\_config:  
 "@type": type.googleapis.com/envoy.config.metrics.v3.HystrixSink  
 num\_buckets: 10

## Contention

The /contention endpoint dumps the current Envoy mutex content stats if mutex tracing is enabled.

## CPU and heap profilers

We can enable or disable the CPU/heap profiler using the /cpuprofiler and /heapprofiler endpoints. Note that this requires compiling Envoy with gperftools. The Envoy GitHub repository has [documentation](https://github.com/envoyproxy/envoy/blob/main/bazel/PPROF.md) on how to do this.

# 7.4 Logging

The /logging endpoint enables or disables different logging levels for a particular component or all loggers.

To list all loggers, we can send a POST request to the /logging endpoint:

$ curl -X POST localhost:9901/logging  
active loggers:  
 admin: info  
 alternate\_protocols\_cache: info  
 aws: info  
 assert: info  
 backtrace: info  
 cache\_filter: info  
 client: info  
 config: info  
...

The output will contain the names of the loggers and the logging level for each logger. To change the logging level for all active loggers, we can use the level parameter. For example, we could run the following to change the logging level of all loggers to debug:

$ curl -X POST localhost:9901/logging?level=debug  
active loggers:  
 admin: debug  
 alternate\_protocols\_cache: debug  
 aws: debug  
 assert: debug  
 backtrace: debug  
 cache\_filter: debug  
 client: debug  
 config: debug  
...

To change a particular logger's level, we can replace the level query parameter name with the logger's name. For example, to change the admin logger level to warning, we can run the following:

$ curl -X POST localhost:9901/logging?admin=warning  
active loggers:  
 admin: warning  
 alternate\_protocols\_cache: info  
 aws: info  
 assert: info  
 backtrace: info  
 cache\_filter: info  
 client: info  
 config: info

To trigger the reopening of all access logs, we can send a POST request to the /reopen\_logs endpoint.

# 7.5 Clusters

The clusters endpoint (/clusters) will show the list of configured clusters and includes the following information:

* Per-host statistics
* Per-host health status
* Circuit breaker settings
* Per-host weight and locality information

Host in this context refers to every discovered host that's part of the upstream clusters.

The snippet below shows what the information looks like (note that the output is trimmed):

{  
 "cluster\_statuses": [  
 {  
 "name": "api\_google\_com",  
 "host\_statuses": [  
 {  
 "address": {  
 "socket\_address": {  
 "address": "10.0.0.1",  
 "port\_value": 8080  
 }  
 },  
 "stats": [  
 {  
 "value": "23",  
 "name": "cx\_total"  
 },  
 {  
 "name": "rq\_error"  
 },  
 {  
 "value": "51",  
 "name": "rq\_success"  
 },  
 ...  
 ],  
 "health\_status": {  
 "eds\_health\_status": "HEALTHY"  
 },  
 "weight": 1,  
 "locality": {}  
 }  
 ],  
 "circuit\_breakers": {  
 "thresholds": [  
 {  
 "max\_connections": 1024,  
 "max\_pending\_requests": 1024,  
 "max\_requests": 1024,  
 "max\_retries": 3  
 },  
 {  
 "priority": "HIGH",  
 "max\_connections": 1024,  
 "max\_pending\_requests": 1024,  
 "max\_requests": 1024,  
 "max\_retries": 3  
 }  
 ]  
 },  
 "observability\_name": "api\_google\_com"  
 },  
 ...

To get the JSON output, we can append the ?format=json when making the request or opening the URL in the browser.

## Host statistics

The output includes the statistics for each host, as explained in the table below:

|  |  |
| --- | --- |
| Metric name | Description |
| cx\_total | Total connections |
| cx\_active | Total active connections |
| cx\_connect\_fail | Total connection failures |
| rq\_total | Total requests |
| rq\_timeout | Total timed out requests |
| rq\_success | Total requests with non-5xx responses |
| rq\_error | Total requests with 5xx responses |
| rq\_active | Total active requests |

## Host health status

The host health status gets reported under the health\_status field. The values in the health status depend on whether the health checking is enabled. Assuming active and passive (circuit breaker) health checking is enabled, the table shows the boolean fields that might be included in the health\_status field.

|  |  |
| --- | --- |
| Field name | Description |
| failed\_active\_health\_check | True, if the host is currently failing active health checks. |
| failed\_outlier\_check | True, if the host is presently considered an outlier and has been ejected. |
| failed\_active\_degraded\_check - True if the host is presently being marked as degraded through active health checking. |  |
| pending\_dynamic\_removal - True if the host has been removed from service discovery but is being stabilized due to active health checking. |  |
| pending\_active\_hc | True, if the host has not yet been health checked. |
| excluded\_via\_immediate\_hc\_fail | True, if the host should be excluded from panic, spillover, etc. calculations because it was explicitly taken out of rotation via protocol signal and is not meant to be routed to. |
| active\_hc\_timeout | True, if the host failed active HC due to timeout. |
| eds\_health\_status | By default, set to healthy (if not using EDS). Otherwise, it can also be set to unhealthy or degraded. |

Note that the fields from the table are reported only if set to true. For example, if the host is healthy, then the health status will look like this:

"health\_status": {  
 "eds\_health\_status": "HEALTHY"  
}

If an active health check is configured and the host is failing, then the status will look like this:

"health\_status": {  
 "failed\_active\_health\_check": true,  
 "eds\_health\_status": "HEALTHY"  
}

# 7.6 Listeners and listener draining

The /listeners endpoints list all configured listeners. This includes the names as well as the address and ports each listener is listening on.

For example:

$ curl localhost:9901/listeners  
http\_8080::0.0.0.0:8080  
http\_hello\_world\_9090::0.0.0.0:9090

For the JSON output, we can append the ?format=json to the URL:

$ curl localhost:9901/listeners?format=json  
{  
 "listener\_statuses": [  
 {  
 "name": "http\_8080",  
 "local\_address": {  
 "socket\_address": {  
 "address": "0.0.0.0",  
 "port\_value": 8080  
 }  
 }  
 },  
 {  
 "name": "http\_hello\_world\_9090",  
 "local\_address": {  
 "socket\_address": {  
 "address": "0.0.0.0",  
 "port\_value": 9090  
 }  
 }  
 }  
 ]  
}

## Listener draining

A typical scenario when draining occurs is during hot restart draining. It involves reducing the number of open connections by instructing the listeners to stop accepting incoming requests before the Envoy process is shut down.

By default, if we shut Envoy down, all connections are immediately closed. To do a graceful shutdown (i.e., don't close existing connections), we can use the /drain\_listeners endpoint with an optional graceful query parameter.

Envoy drains the connection based on the configuration specified through the --drain-time-s and --drain-strategy.

If not provided, the drain time defaults to 10 minutes (600 seconds). The value specifies how long Envoy will drain the connection — i.e., wait before closing them.

The drain strategy parameter determines the behavior during the drain sequence (e.g., during hot restart) where connections are terminated by sending the "Connection: CLOSE" (HTTP/1.1) or GOAWAY frame (HTTP/2).

There are two supported strategies: gradual (default) and immediate. When using the gradual strategy, the percentage of requests encouraged to drain increases to 100% as the drain time elapses. The immediate strategy will enable all requests to drain as soon as the drain sequence begins.

The draining is done per listener. However, it must be supported at the network filter level. The filters that currently support graceful draining are Redis, Mongo, and the HTTP connection manager.

Another option on the endpoint is the ability to drain all inbound listeners using the inboundonly query parameter (e.g., /drain\_listeners?inboundonly). This uses the traffic\_direction field on the listener to determine the traffic direction.

# 7.7 Tap filter

The purpose of the tap filter is to record HTTP traffic based on some matching properties. There are two ways to configure the tap filter: (1) Using the static\_config field inside the Envoy configuration or (2) using the admin\_config field and specifying the configuration ID. The difference is that we provide everything at once in the static configuration — the match configuration and the output configuration. When using the admin configuration, we provide only the configuration ID and then use the /tap administrative endpoint to configure the filter at runtime.

As we alluded to, the filter configuration is separated into two parts: the **match configuration** and the **output configuration**.

We can specify the matching predicate with the match configuration that tells the tap filter which requests to tap and write to the configured output.

For example, the snippet below shows how to use the any\_match to match all requests, regardless of their properties:

common\_config:  
 static\_config:  
 match:  
 any\_match: true  
...

We also have an option to match on request and response headers, trailers, and body.

## Header/trailer match

The header/trailer matchers use the HttpHeadersMatch proto, where we specify an array of headers to match on. For example, this snippet matches any requests where the request header my-header is set precisely to hello.

common\_config:  
 static\_config:  
 match:  
 http\_request\_headers\_match:  
 headers:  
 name: "my-header"  
 string\_match:  
 exact: "hello"  
...

Note that within the string\_match we can use the other matchers (e.g., prefix, suffix, safe\_regex) as explained earlier.

## Body match

The generic request and response body match uses the HttpGenericBodyMatch to specify a string or binary match. As the name suggests, the string match (string\_match) looks for a string within the HTTP body, and the binary match (binary\_match) looks for a sequence of bytes to be located in the HTTP body.

For example, the following snippet matches if the response body contains the string hello:

common\_config:  
 static\_config:  
 match:  
 http\_response\_generic\_body\_match:  
 patterns:  
 string\_match: "hello"  
...

## Match predicates

We can combine multiple headers, trailers, and body matchers with match predicates such as or\_match, and\_match, and not\_match.

The or\_match and and\_match use the MatchSet proto that describes either a logical OR or a logical AND. We specify a list of rules that make up a set in the rules field within the match set.

The example below shows how to use the and\_match to ensure that both the response body contains the word hello and the request header my-header is set to hello:

common\_config:  
 static\_config:  
 match:  
 and\_match:  
 rules:  
 - http\_response\_generic\_body\_match:  
 patterns:  
 - string\_match: "hello"  
 - http\_request\_headers\_match:  
 headers:  
 name: "my-header"  
 string\_match:  
 exact: "hello"  
...

If we wanted to implement the logical OR, then we could replace the and\_match field with the or\_match field. The configuration within the field would stay the same, as both fields use the MatchSet proto.

Let's use the same example as that used previously to show how the not\_match works. Let's say we want to tap all requests that don't have the header my-header: hello set, and for which the response body doesn't include the string hello.

Here's how we could write that configuration:

common\_config:  
 static\_config:  
 match:  
 not\_match:  
 and\_match:  
 rules:  
 - http\_response\_generic\_body\_match:  
 patterns:  
 - string\_match: "hello"  
 - http\_request\_headers\_match:  
 headers:  
 name: "my-header"  
 string\_match:  
 exact: "hello"  
...

The not\_match field uses the MatchPredicate proto just like the parent match field. The match field is a recursive structure, and it allows us to create complex nested match configurations.

The last field to mention here is the any\_match. This is a Boolean field that, when set to true, will always match.

## Output configuration

Once the requests are tapped, we need to tell the filter where to write the output. At the moment, we can configure a single output sink.

Here's how a sample output configuration would look:

...  
output\_config:  
 sinks:  
 - format: JSON\_BODY\_AS\_STRING  
 file\_per\_tap:  
 path\_prefix: tap  
...

Using the file\_per\_tap, we specify that we want to output a single file for every tapped stream. The path\_prefix specifies the prefix for the output file. The files are named using the following format:

<path\_prefix>\_<id>.<pb | json>

The id represents an identifier that allows us to distinguish the recorded trace for stream instances. The file extension (pb or json) depends on the format selection.

The second option for capturing the output is to use the streaming\_admin field. This specifies that the /tap admin endpoint will stream the tapped output. Note that to use the /tap admin endpoint for the output, the tap filter must also be configured using the admin\_config field. If we statically configure the tap filter, we won't use the /tap endpoint to get the output.

### Format selection

We have multiple options for the output format that specifies how messages are written. Let's look at the different formats, starting with the default format, JSON\_BODY\_AS\_BYTES.

The JSON\_BODY\_AS\_BYTES output format outputs the messages as JSON, and any response body data will be in the as\_bytes field that contains the base64 encoded string.

For example, here's how tapped output would look:

{  
 "http\_buffered\_trace": {  
 "request": {  
 "headers": [  
 {  
 "key": ":authority",  
 "value": "localhost:10000"  
 },  
 {  
 "key": ":path",  
 "value": "/"  
 },  
 {  
 "key": ":method",  
 "value": "GET"  
 },  
 {  
 "key": ":scheme",  
 "value": "http"  
 },  
 {  
 "key": "user-agent",  
 "value": "curl/7.64.0"  
 },  
 {  
 "key": "accept",  
 "value": "\*/\*"  
 },  
 {  
 "key": "my-header",  
 "value": "hello"  
 },  
 {  
 "key": "x-forwarded-proto",  
 "value": "http"  
 },  
 {  
 "key": "x-request-id",  
 "value": "67e3e8ac-429a-42fb-945b-ec25927fdcc1"  
 }  
 ],  
 "trailers": []  
 },  
 "response": {  
 "headers": [  
 {  
 "key": ":status",  
 "value": "200"  
 },  
 {  
 "key": "content-length",  
 "value": "5"  
 },  
 {  
 "key": "content-type",  
 "value": "text/plain"  
 },  
 {  
 "key": "date",  
 "value": "Mon, 29 Nov 2021 19:31:43 GMT"  
 },  
 {  
 "key": "server",  
 "value": "envoy"  
 }  
 ],  
 "body": {  
 "truncated": false,  
 "as\_bytes": "aGVsbG8="  
 },  
 "trailers": []  
 }  
 }  
}

Note the as\_bytes field in the body. The value is a base64 encoded representation of the body data (hello in this example).

The second output format is JSON\_BODY\_AS\_STRING. The difference between the previous format is that with JSON\_BODY\_AS\_STRING, the body data is written in the as\_string field as a string. This format is useful when we know that the body is human readable and there's no need to base64 encode the data.

...  
 "body": {  
 "truncated": false,  
 "as\_string": "hello"  
 },  
...

The other three format types are PROTO\_BINARY, PROTO\_BINARY\_LENGTH\_DELIMITED, and PROTO\_TEXT.

The PROTO\_BINARY format writes the output in the binary proto format. This format is not self-delimiting, which means that if the sink writes multiple binary messages without any length information, the data stream will not be useful. If we're writing one message per file, then the output format will be easier to parse.

We can also use the PROTO\_BINARY\_LENGTH\_DELIMITED format, in which messages are written as sequence tuples. Each tuple is the message length (encoded as 32-bit protobuf varint type), followed by the binary message.

Lastly, we can also use the PROTO\_TEXT format, in which the output is written in the protobuf format below.

http\_buffered\_trace {  
 request {  
 headers {  
 key: ":authority"  
 value: "localhost:10000"  
 }  
 headers {  
 key: ":path"  
 value: "/"  
 }  
 headers {  
 key: ":method"  
 value: "GET"  
 }  
 headers {  
 key: ":scheme"  
 value: "http"  
 }  
 headers {  
 key: "user-agent"  
 value: "curl/7.64.0"  
 }  
 headers {  
 key: "accept"  
 value: "\*/\*"  
 }  
 headers {  
 key: "debug"  
 value: "true"  
 }  
 headers {  
 key: "x-forwarded-proto"  
 value: "http"  
 }  
 headers {  
 key: "x-request-id"  
 value: "af6e0879-e057-4efc-83e4-846ff4d46efe"  
 }  
 }  
 response {  
 headers {  
 key: ":status"  
 value: "500"  
 }  
 headers {  
 key: "content-length"  
 value: "5"  
 }  
 headers {  
 key: "content-type"  
 value: "text/plain"  
 }  
 headers {  
 key: "date"  
 value: "Mon, 29 Nov 2021 22:32:40 GMT"  
 }  
 headers {  
 key: "server"  
 value: "envoy"  
 }  
 body {  
 as\_bytes: "hello"  
 }  
 }  
}

## Configuring the tap filter statically

We combine the matching config with the output config (using the file\_per\_tap field) to configure the tap filter statically.

Here's a snippet that configures the tap filter via static configuration:

- name: envoy.filters.http.tap  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.tap.v3.Tap  
 common\_config:  
 static\_config:  
 match\_config:  
 any\_match: true  
 output\_config:  
 sinks:  
 - format: JSON\_BODY\_AS\_STRING  
 file\_per\_tap:  
 path\_prefx: my-tap

The above configuration will match all requests and write the output to file names with my-tap prefix.

## Configuring the tap filter using the /tap endpoint

To use the /tap endpoint, we have to specify the admin\_config and the config\_id in the tap filter configuration:

- name: envoy.filters.http.tap  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.tap.v3.Tap  
 common\_config:  
 admin\_config:  
 config\_id: my\_tap\_config\_id

Once specified, we can send a POST request to the /tap endpoint to configure the tap filter. For example, here's the POST body that configures the tap filter referenced by the my\_tap\_config\_id name:

config\_id: my\_tap\_config\_id  
tap\_config:  
 match\_config:  
 any\_match: true  
 output\_config:  
 sinks:  
 - streaming\_admin: {}

The format in which we specify the match configuration is equivalent to how we set it for the statically provided configuration.

The clear advantage of using the admin configuration and the /tap endpoint is that we can update the match configuration at runtime, and we don't need to restart the Envoy proxy.

# 7.8 Health checks

The /healthcheck/fail can be used to fail inbound health checks. The endpoint /healthcheck/ok is used to revert the effects of the fail endpoint.

Both endpoints require the use of the HTTP health check filter. We might use this for draining the server before shutting it down or when doing complete restarts. When the fail health check option is invoked, all health checks will fail, regardless of their configuration.

# 8.0 Extending Envoy

In this module, we'll learn about different approaches for extending Envoy.

We'll go into more detail about using Lua and Wasm filters to extend Envoy's functionality.

By the end of this module, you'll understand the different ways to extend Envoy and how to use Lua and Wasm filters.

# 8.1 Extensibility Overview

One way to extend Envoy is by implementing different filters that process or augment the requests. These filters can generate statistics, translate protocols, modify the requests, and so on.

An example of filters is HTTP filters, such as the external authz filter and other filters built into the Envoy binary.

Additionally, we can also write our filters that Envoy dynamically loads and runs. We can decide where we want to run the filter in the filter chain by declaring it in the correct order.

We have a couple of options for extending Envoy. By default, Envoy filters are written in C++. However, we can write them in Lua script or use WebAssembly (WASM) to develop Envoy filters in other programming languages.

Note that the Lua and Wasm filters are limited in their APIs compared to the C++ filters.

1. Native C++ API

The first option is to write native C++ filters and then package them with Envoy. This would require us to recompile Envoy and maintain our version of it. Taking this route makes sense if we're trying to solve complex or high-performance use cases.

1. Lua filter

The second option is using the Lua script. There is an HTTP filter in Envoy that allows us to define a Lua script either inline or as an external file and execute it during both the request and response flows.

1. Wasm filter

The last option is Wasm-based filters. We write the filter as a separate Wasm module with this option, and Envoy loads it dynamically at run time.

In the upcoming modules, we'll learn more about the Lua and Wasm filters.

# 8.2 Lua filter

Envoy features a built-in HTTP Lua filter that allows running Lua scripts during request and response flows. Lua is an embeddable scripting language, popular primarily within embedded systems and games. Envoy uses [LuaJIT](https://luajit.org/) (Just-In-Time compiler for Lua) as the runtime. The highest Lua script version supported by the LuaJIT is 5.1, with some features from 5.2.

At runtime, Envoy creates a Lua environment for each worker thread. Because of this, there is no genuinely global data. Any globals created and populated at load time are visible from each worker thread in isolation.

Lua scripts are run as coroutines in a synchronous style, even though they may perform complex asynchronous tasks. This makes it easier to write. Envoy performs all network/async processing via a set of APIs. When an async task is invoked, Envoy suspends the execution of the script and then resumes once the async operation completes.

We shouldn't be performing any blocking operations from scripts, as that would impact Envoys' performance. We should use only Envoy APIs for all IO operations.

We can modify and/or inspect request and response headers, body, and trailers using a Lua script. We can also make outbound async HTTP calls to an upstream host or perform a direct response and skip any further filter iteration. For example, within the Lua script, we can make an upstream HTTP call and directly respond without continuing the execution of other filters.

## How to configure the Lua filter

Lua scripts can be defined inline using the inline\_code field or by referencing a local file using the source\_codes field on the filter:

name: envoy.filters.http.lua  
typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua  
 inline\_code: |  
 -- Called on the request path.  
 function envoy\_on\_request(request\_handle)  
 -- Do something.  
 end  
 -- Called on the response path.  
 function envoy\_on\_response(response\_handle)  
 -- Do something.  
 end  
 source\_codes:  
 myscript.lua:  
 filename: /scripts/myscript.lua

Envoy treats the above script as a global script, and it executes it for every HTTP request. Two global functions can be defined in each script:

function envoy\_on\_request(request\_handle)  
end

and

function envoy\_on\_response(response\_handle)  
end

The envoy\_on\_request function is called on the request path, and the envoy\_on\_response script is called on the response path. Each function receives a handle that has different methods defined. The script can contain either the response or request function, or both.

We also have an option for disabling or overwriting the scripts on a per-route basis on the virtual host, route, or weighted cluster level.

Disabling or referring to an existing Lua script on host, route, or weighted cluster level is done using the typed\_per\_filter\_config field. For example, here's how to refer to an existing script (e.g. some-script.lua) using the typed\_per\_filter\_config:

typed\_per\_filter\_config:  
 envoy.filters.http.lua:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.lua.v3.LuaPerRoute  
 name: some-script.lua

Similarly, instead of specifying the name field, we could define the source\_code and the inline\_string field like this:

typed\_per\_filter\_config:  
 envoy.filters.http.lua:  
 "@type": type.googleapis.com/envoy.extensions.filters.http.lua.v3.LuaPerRoute  
 source\_code:  
 inline\_string: |  
 function envoy\_on\_response(response\_handle)  
 -- Do something on response.  
 end

## Stream handle API

We mentioned earlier that the request\_handle and response\_handle stream handles get passed to the global request and response functions.

The methods available on the stream handle include methods such as headers, body, metadata, various log methods (e.g. logTrace, logInfo, logDebug, ...), httpCall, connection, and more. You can find the full list of methods in the [Lua filter source code](https://github.com/envoyproxy/envoy/blob/d79a3ab49f1aa522d0a465385425e3e00c8db147/source/extensions/filters/http/lua/lua_filter.h#L151).

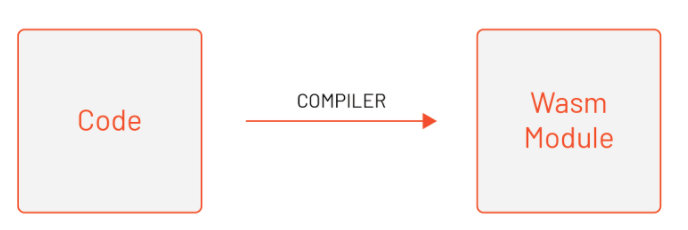
In addition to the stream object, the API supports the following objects:

* [Header object](https://github.com/envoyproxy/envoy/blob/55fc06b43082064cf7551d8dbc08a0e30e2c2f40/source/extensions/filters/http/lua/wrappers.h#L46) (returned by the headers() method)
* Buffer object (returned by the body() method)
* [Dynamic metadata object](https://github.com/envoyproxy/envoy/blob/55fc06b43082064cf7551d8dbc08a0e30e2c2f40/source/extensions/filters/http/lua/wrappers.h#L151) (returned by the metadata() method)
* [Stream info object](https://github.com/envoyproxy/envoy/blob/55fc06b43082064cf7551d8dbc08a0e30e2c2f40/source/extensions/filters/http/lua/wrappers.h#L199) (returned by the streamInfo() method)
* Connection object (returned bt the connection() method)
* [SSL connection info object](https://github.com/envoyproxy/envoy/blob/0fae6970ddaf93f024908ba304bbd2b34e997a51/source/extensions/filters/common/lua/wrappers.h#L124) (returned by the ssl() method on the connection object)

We'll see how to use some of the objects and methods in the Lua lab.

## 8.3 WebAssembly (Wasm)

Wasm is a portable binary format for executable code that relies on an open standard. It allows developers to write in their preferred programming language and then compile the code into a **Wasm module**.



The Wasm modules are isolated from the host environment and executed in a memory-safe sandbox called a **virtual machine (VM)**. Wasm modules use an API to communicate with the host environment.

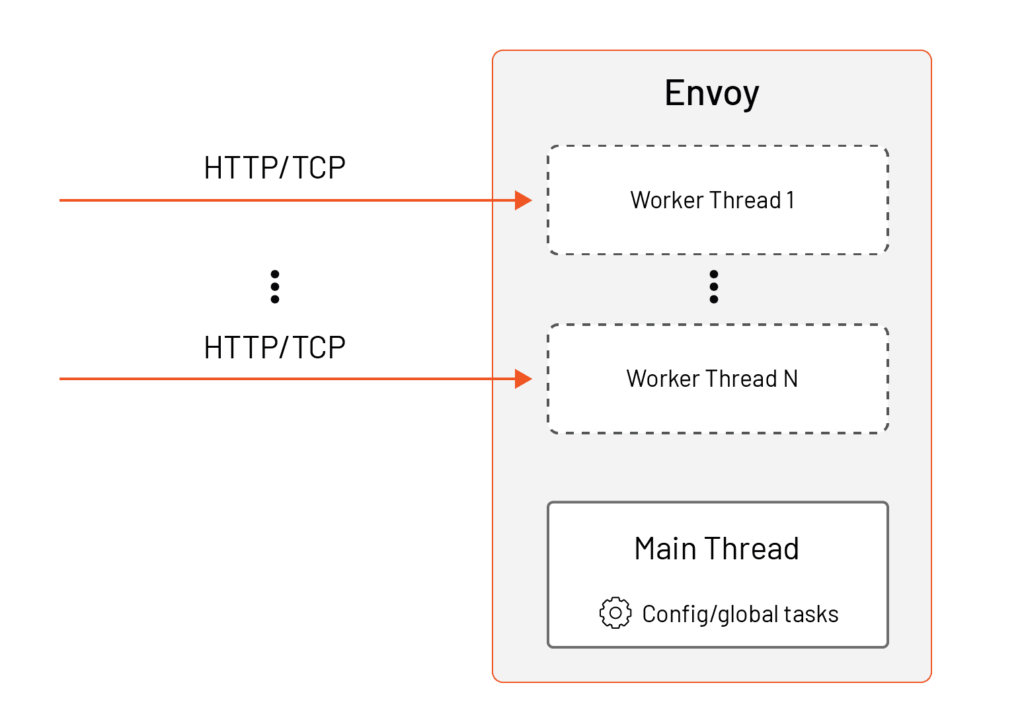
The main goal of Wasm was to enable high-performance applications on web pages. For example, let's say we're building a web application with Javascript. We could write some in Go (or other languages) and compile it into a binary file, the Wasm module. Then, we could run the compiled Wasm module in the same sandbox as the Javascript web application.

Initially, Wasm was designed to run in the web browser. However, we can embed virtual machines into other host applications and execute them. This is what Envoy does!

Envoy embeds a subset of a V8 VM. V8 is a high-performance JavaScript and WebAssembly engine written in C++, and it's used in Chrome and in Node.js, among others.

We mentioned earlier in this course that Envoy operates using a multithreaded model. That means there's one main thread that's responsible for handling configuration updates and executing global tasks.

In addition to the main thread, there are also worker threads responsible for proxying individual HTTP requests and TCP connections. The worker threads are designed to be independent of each other. For example, a worker thread processing one HTTP request will not be impacted by other worker threads processing other requests.



Each thread owns its replica of resources, including the Wasm VMs. The reason for that is to avoid any expensive cross-thread synchronization in terms of higher memory usage.

Envoy loads every unique Wasm module (all \*.wasm files) into a unique Wasm VM at run time. Since Wasm VM is not thread-safe (i.e., multiple threads would have to synchronize access to a single Wasm VM), Envoy creates a separate replica of Wasm VM for every thread on which the extension will be executed. Consequently, every thread might have multiple Wasm VMs in use at the same time.

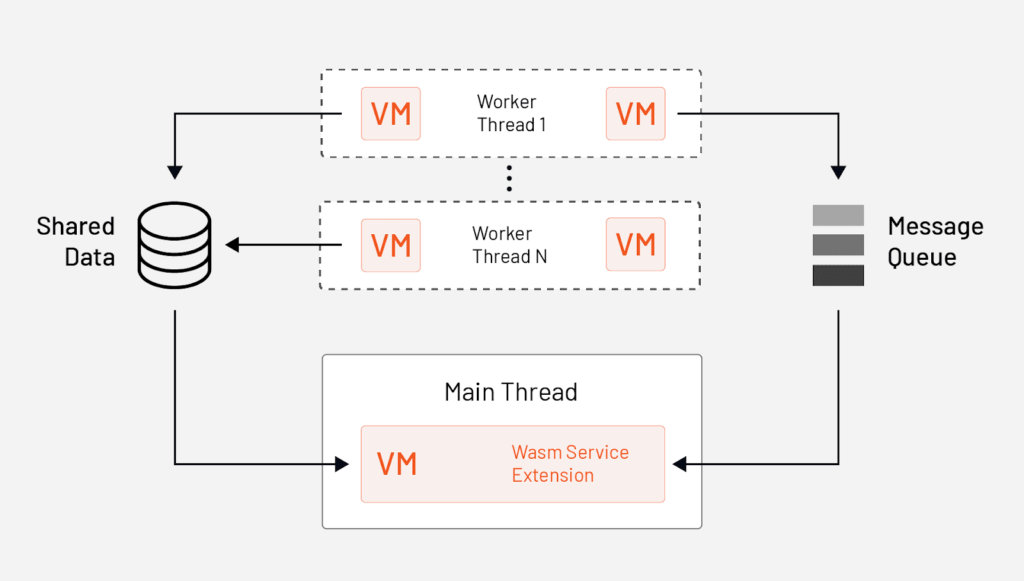
## Proxy-Wasm

The SDK we'll use allows us to write Wasm extensions that are either HTTP filters, Network Filters, or a dedicated extension type called **Wasm Service**. These extensions are executed inside a Wasm VM on a worker thread (HTTP filters, Network filters) or on the main thread (Wasm Service). As we mentioned, the threads are independent, and they are inherently unaware of request processing happening on other threads.

The HTTP filter is the one that handles the HTTP protocol, and it operates on HTTP headers, body, and so on. Similarly, the network filter handles TCP protocols and operates on data frames and connections. We can also say that these two plugin types are stateless.

Envoy also supports stateful scenarios. For example, you could write an extension that aggregates stats such as request data, logs, or metrics across multiple requests — this essentially means across numerous worker threads. For this scenario, we'd use the Wasm Service type. The Wasm service type runs on a singleton VM; there's only one instance of this VM, and it runs on the Envoy main thread. You can use it to aggregate metrics or logs from the stateless filters.

The figure below shows how the Wasm service extension is executed on the main thread, instead of HTTP or network filters, which are executed on worker threads.



The fact that the Wasm service extension is executed on the main thread doesn't impact the request latency. On the other hand, the network or HTTP filters can impact the latency.

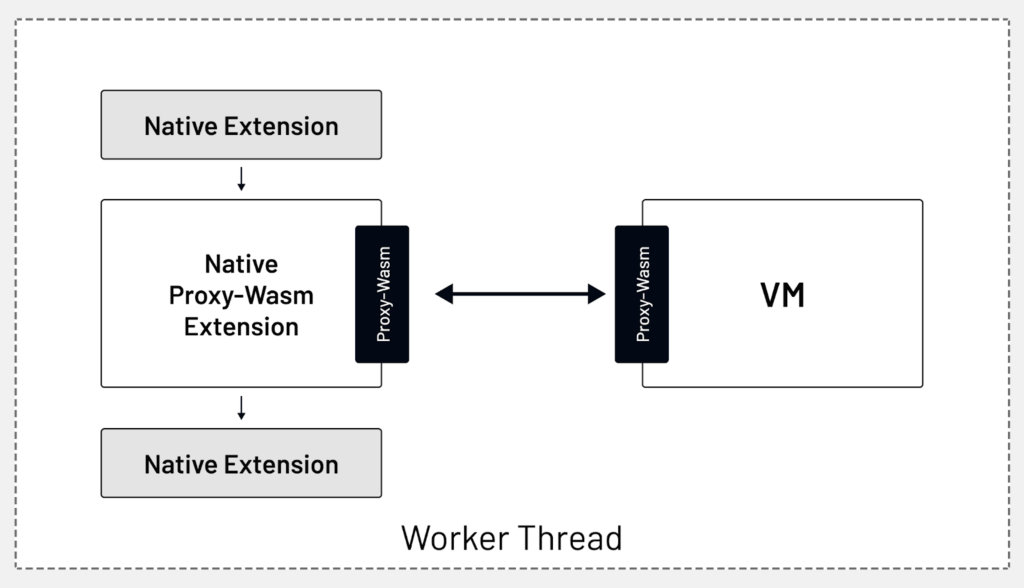
The figure shows a Wasm Service extension running on the main thread that uses the Message Queue API to subscribe to a queue and receive messages sent by the HTTP filter or Network Filter running on the worker threads. The Wasm Service extension can then aggregate data received from the worker threads.

Wasm Service Extensions aren't the only way to persist data. You can also call out to HTTP or gRPC APIs. Moreover, we can perform actions outside requests using the timer API.

The APIs we mentioned, the message queue, timer, and shared data are all defined by a component called [Proxy-Wasm](https://github.com/proxy-wasm).

Proxy-Wasm is a proxy-agnostic ABI (Application Binary Interface) standard that specifies how proxies (our hosts) and the Wasm modules interact. These interactions are implemented in the form of functions and callbacks.

The APIs in Proxy-Wasm are proxy agnostic, which means that they work with Envoy proxies as well as any other proxies ([MOSN](https://github.com/mosn/mosn), for example) that implement the Proxy-Wasm standard. This makes your Wasm filters portable between different proxies, and they aren't tied to Envoy only.



As the requests come into Envoy, they go through different filter chains, get processed by a filter, and at some point in the chain, the request data flows through the native Proxy-Wasm extension.

This extension uses the Proxy-Wasm interface to talk to the extension running inside the VM. Once the filter processes the data, the chain continues, or stops, depending on the result returned from the extension.

Based on the Proxy-Wasm spec, we can use a couple of language-specific SDK implementations to write the extensions in.

In one of the labs, we'll use [Go SDK for Proxy-Wasm](https://github.com/tetratelabs/proxy-wasm-go-sdk) to write Proxy-Wasm plugins in Go.

[TinyGo](https://tinygo.org/) is a compiler used for embedded systems and WebAssembly. It doesn't support the use of all standard Go packages. For example, some standard packages such as net and others are not supported.

You also have the option of using Assembly Script, C++, Rust, or Zig.

## Configuring Wasm extensions

The generic Wasm extension configuration in Envoy looks like this:

- name: envoy.filters.http.wasm  
 typed\_config:  
 "@type": type.googleapis.com/udpa.type.v1.TypedStruct  
 type\_url: type.googleapis.com/envoy.extensions.filters.http.wasm.v3.Wasm  
 value:  
 config:  
 vm\_config:  
 vm\_id: "my\_vm"  
 runtime: "envoy.wasm.runtime.v8"  
 configuration:  
 "@type": type.googleapis.com/google.protobuf.StringValue  
 value: '{"plugin-config": "some-value"}'  
 code:  
 local:  
 filename: "my-plugin.wasm"  
 configuration:  
 "@type": type.googleapis.com/google.protobuf.StringValue  
 value: '{"vm-wide-config": "some-value"}'

The vm\_config field is used to specify the Wasm VM, runtime, and the actual pointer to the .wasm extension we want to execute.

The vm\_id field is used when communicating between VMs. This ID can then be used to share data between VMs through shared data APIs and queues. Note that to reuse the VMs across multiple plugins, you have to use the same vm\_id, runtime, configuration, and code.

The next item is the runtime. This is usually set to envoy.wasm.runtime.v8. For example, if we compile the Wasm extension with the Envoy, we'd use the null runtime here. The other options are Wasm micro runtime, Wasm VM, or Wasmtime; - none of these are enabled in the official Envoy builds, however.

The configuration under the vm\_config field is used to configure the VM itself. In addition to the VM ID and runtime, the other important piece is the code field.

The code field is where we reference the compiled Wasm extension. This can be either a pointer to a local file (e.g., /etc/envoy/my-plugin.wasm) or a remote location (e.g., https://wasm.example.com/my-plugin.wasm).

The configuration files, one under the vm\_config and the other at the config level, are used to provide configuration for the VM and the plugin. These values can then be read from the Wasm extension code when the VM or plugin starts.

To run a Wasm service plugin, we have to define the configuration in the bootstrap\_extensions field and set the singleton Boolean field value to true.

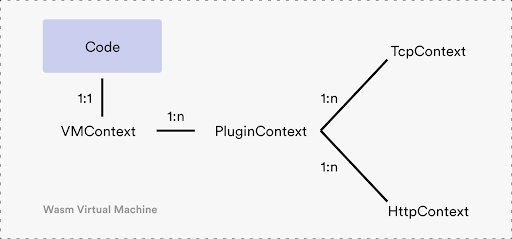
bootstrap\_extensions:  
- name: envoy.bootstrap.wasm  
 typed\_config:  
 "@type": type.googleapis.com/envoy.extensions.wasm.v3.WasmService  
 singleton: true  
 config:  
 vm\_config: { ... }

## Developing Wasm extensions – Proxy-Wasm Go SDK API

When developing Wasm extensions, we'll learn about the contexts, hostcall API, and entry points.

### Contexts

Contexts are a collection of interfaces in the Proxy-Wasm SDK and are mapped to the concepts we explained earlier.



For example, there's a single VMContext in each VM and can have one or more PluginContexts. This means we can run different plugins within the same VM context (i.e., when using the same vm\_id). Each PluginContext corresponds to a plugin instance. That's either a TcpContext (TCP network filter) or HttpContext (HTTP filter).

The VMContext interfaces has two functions defined: the OnVMStart function and the NewPluginContext function:

type VMContext interface {  
 OnVMStart(vmConfigurationSize int) OnVMStartStatus  
 NewPluginContext(contextID uint32) PluginContext  
}

As the names suggest, the OnVMStart is called after the VM is created. Within this function, we can retrieve the optional VM configuration using the GetVMConfiguration hostcall. The purpose of this function is to perform any VM-wide initialization.

As developers, we need to implement the NewPluginContext function where we create an instance of the PluginContext.

The PluginContext interface defines functionally similar functions as the VMContext. Here's the interface:

type PluginContext interface {  
 OnPluginStart(pluginConfigurationSize int) OnPluginStartStatus  
 OnPluginDone() bool  
  
 OnQueueReady(queueID uint32)  
 OnTick()  
  
 NewTcpContext(contextID uint32) TcpContext  
 NewHttpContext(contextID uint32) HttpContext  
}

The OnPluginStart function is comparable to the OnVMStart function we mentioned earlier. It gets called when the plugin gets created. Within this function, we also can retrieve the plugin-specific configuration using the GetPluginConfiguration API. We also have to implement either the NewTcpContext or NewHttpContext that gets called in response to HTTP/TCP streams in the proxy. This context also contains some other functions used for setting up the queue (OnQueueReady) or doing asynchronous tasks in parallel to the stream processing (OnTick).

Refer to the context.go file in the [Proxy Wasm Go SDK Github repository](https://github.com/tetratelabs/proxy-wasm-go-sdk/blob/main/proxywasm/types/context.go) for the latest interface definitions.

### Hostcall API

The hostcall API, implemented [here](https://github.com/tetratelabs/proxy-wasm-go-sdk/blob/main/proxywasm/hostcall.go), gives us ways to interact with the Envoy proxy from the Wasm plugin.

The hostcall API defines methods for reading the configuration; setting up a shared queue and performing queue operations; dispatching HTTP calls, retrieving headers, trailers, and body from both request and response streams and manipulating these values; configuring metrics; and more.

### Entry point

The entry point for the plugins is the main function. Envoy creates the VMs, and before it tries to create the VMContext, it calls the main function. In the typical implementation, we call the SetVMContext method the main function:

func main() {  
 proxywasm.SetVMContext(&myVMContext{})  
}  
  
type myVMContext struct { .... }  
  
var \_ types.VMContext = &myVMContext{}