HAProxy as an API Gateway

Consolidate Your APIs Behind the World's Fastest Software Load Balancer

HAPROXY

HAProxy as an API Gateway

Consolidate Your APIs Behind the World's Fastest Software Load Balancer

Table of Contents

Introduction

An API gateway is simply a reverse proxy or load balancer that sits between client applications and the API services that they call.

An API gateway decouples client-side applications—whether they be web applications, mobile apps, IoT devices, or desktop applications—from the API services that feed them data. Client applications no longer need to connect to API servers directly, which would complicate both the client and server by tying one to the other. Instead, an API gateway mediates their communication.

HAProxy, the world's fastest and most widely used software load balancer, fills the role as an API gateway extremely well. In addition to routing API calls to the proper backend servers, it also handles load balancing, security, rate limiting, caching, monitoring, and other cross-cutting concerns. By placing all of your APIs behind an HAProxy load balancer, you can offload those requirements.

In this book, you will learn how best to utilize HAProxy as an API gateway, and simplify your infrastructure as a result!

 slack.haproxy.org/

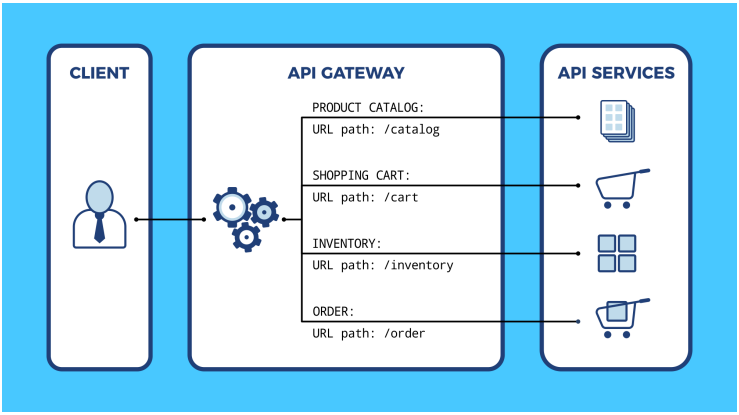
 twitter.com/haproxy

 github.com/haproxy/haproxy

 hub.docker.com/u/haproxytech

Essential Features of an API Gateway

The API gateway becomes the glue layer that connects requests from frontend applications to API endpoints on the backend. It keeps these layers from becoming tightly coupled. Your applications can point to a single destination—the gateway—and it will handle the routing.



Routing is one of the essential features of an API gateway, but there are others that are just as important. An API gateway can handle:

- HTTP routing
- Load balancing
- Server overload protection

- Rate limiting
- Metrics and logging

Let's explore these features in more detail.

HTTP routing

The primary role of the API gateway is to route a client's incoming requests to the appropriate internal service. HAProxy can route based on any information found in the HTTP request including portions of the URL path, query string, and HTTP headers.

In the following example, our configuration sets up a *frontend* that accepts incoming requests on port 443, checks if the URL path is **/cart** or **/catalog**, and then forwards them to the correct backend.

```
frontend api_gateway
  bind :443 ssl crt /etc/haproxy/ssl/cert.pem
  use_backend be_cart if { path_beg -i /cart }
  use_backend be_catalog if { path_beg -i /catalog
}

backend be_cart
  server s1 10.0.0.3:80

backend be_catalog
  server s1 10.0.0.5:80
```

If you need to inspect tens, hundreds, or even thousands of paths in a single route, then it is better to manage them through a *map file*. A map file stores key/value pairs in memory.

In our example, the key would be the path string and the value would be the name of the backend to route the request to. The map file **routing.map** would contain:

```
# endpoint   backend name
/cart       be_cart
/catalog    be_catalog
```

Our HAProxy configuration would contain a *use_backend* line that finds the backend name in the map file based on which line in the file the URL path matches:

```
frontend api_gateway
  # ...
  use_backend
↪ %[path,map_beg("/etc/haproxy/routing.map")]
```

If you assign unique domain names to different APIs, then you can check the *Host* header to determine how to route requests. Below, we route API requests depending on the domain. When accessing **cart.haproxy.com**, it routes to the cart servers and when accessing **catalog.haproxy.com** it routes to the catalog servers:

```
frontend api_gateway
  bind :443 ssl crt /etc/haproxy/ssl/cert.pem

  use_backend be_cart if { req.hdr(Host) -i
↪ -m dom cart.haproxy.com }
  use_backend be_catalog if { req.hdr(Host) -i
↪ -m dom catalog.haproxy.com }

backend be_cart
  server s1 10.0.0.3:80

backend be_catalog
  server s1 10.0.0.5:80
```

HAProxy is extremely flexible and these examples are just simple use cases. You can apply more complex logic for HTTP routing and request handling.

To simplify adding or removing rows to map files across a cluster of HAProxy instances, you can upgrade to HAProxy Enterprise and use the Update module.

Load balancing

You can improve the performance and resilience of your API endpoints by replicating the service across several nodes. Then, the API gateway will balance incoming client requests among them. You can adjust the load balancing algorithm to suit the type of service and protocol.

- For quick and short API calls, use the *roundrobin* algorithm;
- For longer-lived websockets, use the *leastconn* algorithm;
- For services that have backend servers optimized to process particular functions, use the *uri* algorithm, which hashes the URI so that future requests for the same URI go to the same sever.

In the following example, the API backend is balanced across two nodes using the *roundrobin* algorithm.

```
backend cart_api
  balance roundrobin
  server s1 10.0.0.3:80
  server s2 10.0.0.4:80
```

Load balancing your API endpoints improves performance and creates redundancy by sharing load across a pool of servers. You can choose the most appropriate balancing algorithm on a per-backend basis.

You can also define health checks for your servers so that HAProxy automatically reroutes traffic if there's a problem. In the following example, we monitor the health of the servers by sending GET requests to the URL path **/health**:

```
backend mobile_api
  balance roundrobin
  option httpchk GET /health
  server s1 10.0.0.3:80 check
  server s2 10.0.0.4:80 check
```

The *option httpchk* directive sets the method (i.e. GET) and URL to monitor. A *check* parameter on each server line enables the feature.

Server overload protection

The HAProxy load balancer stands in a strategic position, between your clients and services, ensuring that no backend nodes are overloaded by spikes in traffic. Without this, all requests would be forwarded to the backend servers, risking high wait times and timeouts.

HAProxy implements queuing mechanisms to prevent sending too many requests at once to a service. Add the *maxconn* argument to a *server* directive to queue requests in the gateway when the server is already handling 100 connections:

```
backend mobile_api
  balance roundrobin
  server s1 10.0.0.3:80 maxconn 100
  server s2 10.0.0.4:80 maxconn 100
```

In this case, up to 100 connections can be established at once to a server. Any more than that will be queued. This relieves strain on your servers, allowing them to process requests more efficiently.

Rate limiting

You may want to limit the number of requests a client can send to your APIs within a period of time. This might be to enforce a limit depending on the user's API subscription level, for example. To send more requests, clients could subscribe to a higher-priced tier.

In HAProxy, *stick tables*, which are an in-memory data storage, track clients by IP address, URL parameter, cookie, or other HTTP header. In the next example, the client passes a URL parameter called *apitoken*, which we use to count their number of requests. They're limited to 1000 requests within 24 hours. The period is set with the *expire* parameter on the stick-table directive.

```
frontend api_gateway
  bind :443 ssl crt /etc/haproxy/ssl/cert.pem
  stick-table type string size 1m expire 24h
↪ store http_req_cnt

  acl exceeds_limit
↪ url_param(apitoken),table_http_req_cnt gt 1000

  http-request deny deny_status 429 if
exceeds_limit
  http-request track-sc0 url_param(apitoken)
```

Now, as I make requests to the site, passing the URL parameter, **apitoken=mytoken**, the count of HTTP requests is incremented. When clients go past their limit, they receive a *429 Too Many Requests* response. Check out our blog post Introduction to HAProxy Stick Tables for more information about defining stick tables and other examples of rate limiting.

Adding *deny_status* to the *http-request deny* directive allows you to set a custom response code when rejecting requests. Possible values are 200, 400, 403, 405, 408, 425, 429, 500, 502, 503, 504.

Detailed metrics and logs

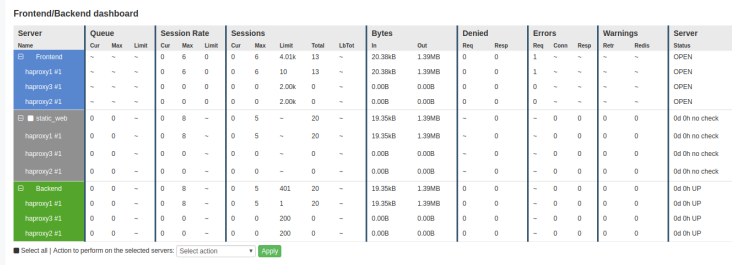
HAProxy is famous for the level of details it provides on the traffic it processes. There are two main features: the statistic dashboard and the logs.

Statistics Dashboard

HAProxy and HAProxy Enterprise provide a statistics page that shows many metrics for each frontend, backend, and server. Enable it by adding a *frontend* section with a *stats enable* directive to your HAProxy configuration file. This will start the HAProxy Stats page on port 8404:

```
frontend stats
  bind *:8404
  stats enable
  stats uri /
  stats refresh 5s
```

If you're running HAProxy Enterprise, then you have access to the Real Time Dashboard, which gives you even more options. The image below shows the HAProxy Enterprise *Real Time Dashboard*:



This data is also available by querying the HAProxy Runtime API, which can return results as JSON or CSV,

making it easy to integrate with third-party monitoring tools.

There's also a built-in Prometheus exporter. The image below shows an integration of HAProxy with Grafana, via the Prometheus exporter:

Logs

The HAProxy access logs are a trove of information, since HAProxy can report the following information for each API call:

- Client IP and port
- Routing info: frontend, backend, server selected
- Requested URL with query string
- Timers: time for the client to send the request, server connection time, response time, total session duration, etc...
- Termination status: did the session finished properly or not? If not what happened and at what phase of the session?
- Any custom header or cookie you want to capture
- SSL / TLS information

Based on all the information it provides, it's possible to build reports that help to identify quickly where problems are occurring (e.g. Is it networking? The application? A particular server?).

Conclusion

In this chapter, you learned some of the essential features of an API gateway. The HAProxy load balancer works as an API gateway: managing routing, load balancing, server overload protection, rate limiting and logging/metrics.

Health Checks

HAProxy monitors the health of your servers and removes them from the load balancing rotation if they go offline or return errors, helping you to meet the service-level objectives you've set for your API services.

In this chapter, you will learn several ways to configure health checks to create highly available services using HAProxy as an API gateway.

Active health checks

The easiest way to detect whether a server is up is with an *active check*. HAProxy polls the server on a fixed interval by trying to make a TCP connection. If it can't connect, the check fails and that server is removed from the load balancing rotation.

Consider the following example:

```
backend apiservers
  balance roundrobin
  server server1 192.168.50.3:80 check
```

The *check* parameter on the *server* line enables active health checking for that server. Once you've enabled active checking, you can then add other parameters to change the polling interval, the number of failed checks that

trigger an action, and how many checks the server must pass to be brought back online.

The next example demonstrates these parameters:

```
server server1 192.168.50.3:80 check inter 5s
↪ downinter 5s fall 3 rise 3
```

The *inter* parameter sets the interval between checks. It defaults to two seconds. The *downinter* parameter sets the interval between checks when the server is already in a down state. The *fall* parameter sets the number of failed checks allowed before marking the server as down and *rise* sets how many successful checks there must be before marking the server as up again.

For web applications, instead of basing the checks on whether you can make a TCP connection, you can send an HTTP request instead. Add *option httpchk* to the *backend* section; HAProxy will send a simple HTTP request and expect to receive a successful HTTP response:

```
backend apiservers
  balance roundrobin
  option httpchk GET /check
  server server1 192.168.50.3:80 check
```

The *option httpchk* directive lets you choose the HTTP method (e.g. GET), as well as the URL path to monitor (e.g. **/check**). Having this flexibility means that you can dedicate a specific webpage to be the health-check endpoint.

You can accept only specific responses from the server too, such as a specific status code or a string within the HTTP body. Use *http-check expect* with either the *status* or *string* keyword. In the following example, only health checks that return a 200 OK response are classified as successful:

```
backend apiservers
  balance roundrobin
  option httpchk GET /check
  http-check expect status 200
  server server1 192.168.50.3:80 check
```

Or, require that the response body contain a certain case-sensitive string of text. Below, we expect the body of the response to contain the string *success*.

```
http-check expect string success
```

Passive health checks

Active checks are easy to configure and provide a simple monitoring strategy. They work well in a lot of cases, but they have one shortcoming: They don't catch errors that affect parts of the application not in the direct line of sight of the health check. For example, your health check may poll the URL path **/check**, but not other web pages.

Passive checks monitor all traffic for errors. Configure passive health checks by by adding an *observe* parameter to a *server* line. In the following example, the *observe* parameter has a value of *layer4*, which means that it watches all connections from HAProxy to the backend server for failures:

```
backend apiservers
  balance roundrobin
  option redispatch
  retries 3
  timeout connect 4s
  server server1 192.168.50.3:80 check inter 2m
↪ downinter 2m observe layer4 error-limit 10
↪ on-error mark-down
```

Here, we're observing all TCP connections for problems. We've set *error-limit* to 10, which means that ten connections must fail before the *on-error* parameter is invoked and marks the server as down. When that happens, you'll see a message like this in the HAProxy logs:

Server apiservers/server1 is DOWN, reason: Health analyze, info: "Detected 10 consecutive errors, last one was: L4 unsuccessful connection". 1 active and 0 backup servers left. 0 sessions active, 0 requeued, 0 remaining in queue.

The only way to revive the server is with the regular health checks. In this example, we've used *inter* to set the active health check interval to two minutes. When they mark the

server as healthy again, you'll see a message telling you that the server is back up:

Server apiservers/server1 is UP, reason: Layer4 check passed, check duration: 0ms. 2 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.

It's a good idea to include *option redispatch* so that if a client runs into an error connecting, they'll be redirected to another healthy server instantly. They'll never know that there was an issue. You can also add a *retries* parameter so that HAProxy tries to connect the given number of times. The delay between retries is set with *timeout connect*.

When you set *observe layer4*, HAProxy monitors traffic for failed connections. You can also monitor for failed HTTP requests by setting *observe layer7*. This will automatically remove servers if users experience HTTP errors.

```
server server1 192.168.50.3:80 check inter 2m
↪ downinter 2m observe layer7 error-limit 10
↪ on-error mark-down
```

If any webpage returns a status code other than 100-499, 501 or 505, it will count towards the error limit. Beware, however, that if enough users are getting errors on even a single, misconfigured webpage, it could cause the entire server to be removed.

Agent health checks

Active and passive checks will give you a good indication of how your application servers are functioning, instantly removing bad nodes from load balancing. One downside though is that they don't give you a rich sense of the server's state, such as its CPU load, free disk space, and network throughput.

With HAProxy, you can query agent software that's running on the server itself. Since the agent has full access to the remote server, it has the ability to check its vitals more closely.

Agents have an edge over other types of health checks. They can send signals back to HAProxy to force a change in state. For example, they can mark the server as up or down, put it into maintenance mode, change the percentage of traffic flowing to it, and increase or decrease the maximum number of concurrent connections allowed. The agent will trigger your chosen action when some condition occurs, such as when CPU usage spikes or disk space runs low.

Consider this example that configures HAProxy to communicate with a remote agent listening at port 9999 on the server:

```
backend apiservers
  balance roundrobin
  server server1 192.168.50.3:80 check weight 100
↪ agent-check agent-inter 5s agent-addr
↪ 192.168.50.3 agent-port 9999
```

The *server* directive's *agent-check* parameter tells HAProxy to connect to an external agent; The *agent-addr* and *agent-port* parameters set the agent's IP address and port; The interval between checks is set with *agent-inter*.

Note that this communication is not HTTP, but rather a raw TCP connection over which the agent communicates back to HAProxy by sending ASCII text. Here are a few things that it might send back. Notice that an end-of-line character (e.g. \n) is required after the message:

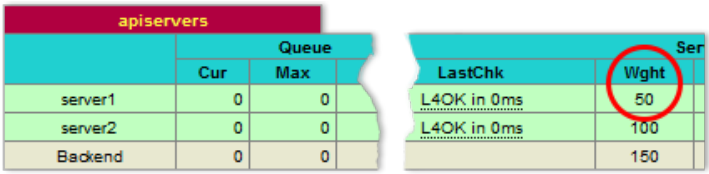
Sends back	**Result**
down\n	server is put into the down state
up\n	server is put into the up state
maint\n	server is put into maintenance mode
ready\n	server is taken out of maintenance mode
50%\n	server's weight is halved
maxconn:10\n	server's maxconn is set to 10

The agent can be any custom-written software that has the ability to return a string when HAProxy connects to it. The following *Go* code creates a TCP server that listens on port 9999, measures the current CPU idle time, and if that metric falls below 10 sends back the string 50%\n, which sets the server's *weight* in HAProxy to half.

```
package main

import (
  "fmt"
  "time"
  "github.com/firstrow/tcp_server"
  "github.com/mackerelio/go-osstat/cpu"
)

func main() {
  server := tcp_server.New(":9999")

  server.OnNewClient(func(c *tcp_server.Client) {
    fmt.Println("Client connected")
    cpuIdle, err := getIdleTime()

    if err != nil {
      fmt.Println(err)
      c.Close()
      return
    }

    if cpuIdle < 10 {
      // Set server weight to half
      c.Send("50%\n")
    } else {
      c.Send("100%\n")
    }

    c.Close()
  })
```

```
  server.Listen()
}

func getIdleTime() (float64, error) {
  before, err := cpu.Get()

  if err != nil {
    return 0, err
  }

  time.Sleep(time.Duration(1) * time.Second)
  after, err := cpu.Get()

  if err != nil {
    return 0, err
  }

  total := float64(after.Total - before.Total)
  cpuIdle := float64(after.Idle-before.Idle) /
↪ total * 100
  return cpuIdle, nil
}
```

To test it out, you can artificially spike the CPU with a tool like stress. Use the HAProxy Stats page to see the effect on the load balancer. Here, the server's weight began at 100, but is set to 50 when there is high CPU usage.



apiservers	Queue			Ser	
	Cur	Max	LastChk	Wght	
server1	0	0	L4OK in 0ms	50	
server2	0	0	L4OK in 0ms	100	
Backend	0	0		150	

Look for the *Weight* column to assess the effects of your stress test.

Conclusion

In this chapter, you learned various ways to health check your servers so that your APIs maintain a high level of reliability. This includes active, passive, and agent-based health checks.

Metrics

HAProxy publishes more than 100 metrics about the traffic flowing through it. When you use HAProxy as an API gateway, these give you insight into how clients are accessing your APIs. Several metrics come to mind as particularly useful, since they can help you determine whether you're meeting your service-level objectives and can detect issues with your services early on.

Let's discuss several that might come in handy.

Important API metrics

Are your API servers up? Keeping an eye on *server health status* is critical for knowing how many servers are passing the health-check probes that HAProxy sends. HAProxy publishes the up/down status of every server along with the pass/fail result of the most recent health check. You could use this to know when more than 25% of your servers are down, for example.

How often are clients calling your API's functions? HAProxy records *request rate*, which is great for seeing usage trends. Knowing which services and functions are the most popular could help when deciding where to add new features or increase server capacity. Use it to reveal unusual traffic patterns too, such as malicious activity like DDoS attacks and faulty client-side code that may be invoking a function repeatedly. HAProxy can enforce rate

limits to protect your servers. Detecting client-side code that's gone haywire might be important if you sell access to your APIs and tie the price to the number of calls a client makes.

Another important metric is the *number of errors*. When a response from a server travels back through HAProxy on its way to the client, we get its status code. For example, statuses in the 400-499 range indicate client-side errors and those in the 500-599 range indicate server-side errors. A relatively small number of client-side errors may indicate only a single misconfigured client, but a sharp increase may be cause for concern; It may be due to a client intentionally trying to abuse your service. Server-side errors generally trace back to bugs introduced during the last deployment.

Also keep an eye on *average response time*, since it shows how snappy your APIs seem to clients. A slowdown may be related to a slow database query or a sudden surge of requests overwhelming your servers. Of course, HAProxy can queue requests before they reach your servers so that your servers always operate within the ideal range of traffic volume.

A final counter to monitor closely is the *number of retries*. HAProxy has the ability to retry a failed connection or HTTP request. It can retry with the same server or, if the redispatch option is set, retry with a different server. If the second, third, or even subsequent try succeeds you won't see an error status in your HAProxy metrics, but you will see your number of retries increase. A large number of retries might implicate your network as the culprit.

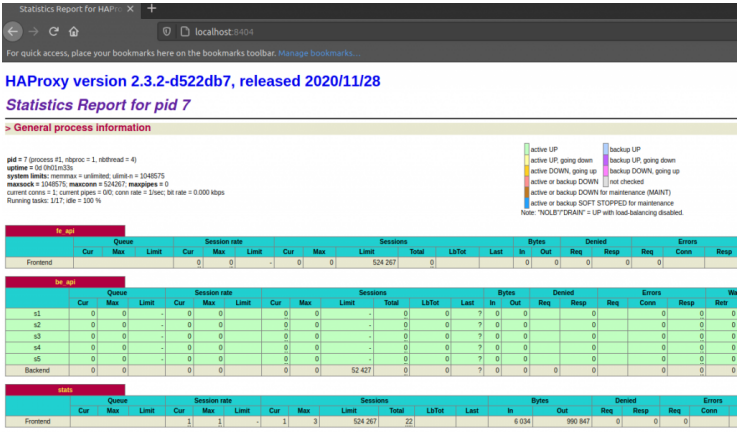
How HAProxy publishes metrics

As HAProxy has evolved, it has included more and more ways to extract its metrics. If you're looking for a quick way to see current numbers, enable the built-in HAProxy Stats page. There you'll find more than 100 unique metrics. Although it doesn't store historical data, it's convenient for checking server health, current request rate, error rate, and more without any other monitoring software.

Enable the Stats page by adding this frontend section to your HAProxy configuration file:

```
frontend stats
  bind :8404
  stats enable
  stats uri /
  stats refresh 10s
```

The dashboard then runs on port 8404:



Learn about the fields shown on the Stats page in our blog post [Exploring the HAProxy Stats Page](#).

You can also fetch the same metrics in a more programmatic way by using the [HAProxy Runtime API](#). First, enable the API by adding a *stats socket* directive to the *global* section of your configuration. This exposes the API as a Unix socket located at **/var/run/haproxy.sock** so you can call it from scripts and programs running on the same machine:

```
global
  stats socket /var/run/haproxy.sock user haproxy
↪ group haproxy mode 660 level admin
```

You can also publish it on an IP address and port of your choosing so that you can access it remotely. In the

following example, the API listens at **localhost** on port 9999:

```
global
  stats socket ipv4@127.0.0.1:9999 user haproxy
↪ group haproxy mode 660 level admin
```

Assuming you're using the IP address, send the *show stat* command to the API by using the *socat* program. You'll get the metrics in CSV format by default, but you can change this to JSON by passing the *json* parameter. Piping the JSON results to Python's *json.tool* program formats the output in a human-readable way.

```
$ echo "show stat json" |\
   socat tcp-connect:127.0.0.1:9999 - |\
   python3 -m json.tool
```

You can also pipe the results to the *cut* and *column* commands to display only the data you want to see. Here, we get the up/down status, request rate, number of errors, average response time and number of retries for the servers listed in a backend named *be_api*. The *watch* command updates the numbers every two seconds:

```
$ watch 'echo "show stat" |\
   socat tcp-connect:127.0.0.1:9999 - |\
   cut -d "," -f 1-2,16,18,43,44,47,61 |\
   column -s, -t'

# pxname  svname        wretr  status  hrsp_4xx
hrsp_5xx  req_rate  rtime
fe_api    FRONTEND  OPEN   0          0
10
be_api    s1        0      UP         0
0         4
be_api    s2        0      UP         0
0         4
be_api    s3        0      UP         0
0         5
be_api    s4        0      UP         0
0         5
be_api    s5        0      UP         0
0         5
be_api    BACKEND    0      UP         0
0         4
```

A third way to fetch metrics from HAProxy is through its integrated Prometheus exporter. Prometheus is an open-source tool for collecting and storing time-series data. Applications that want to publish Prometheus metrics host a webpage, usually at the URL **/metrics**, that a Prometheus server will scrape at an interval. You'll see how to set this up in the next section.

If you use HAProxy Enterprise, you have access to the Send Metrics module too. This module lets you define a custom format for your metrics and then stream them to any URL that you choose. This makes it possible to integrate HAProxy Enterprise with nearly any observability platform.

Visualizing the metrics

When it comes to visualizing HAProxy's metrics over time, there are many options—both free and commercial. To get you started, I'll demonstrate how to set up two open-source graphing tools: Grafana from Grafana Labs and Kibana from Elastic.

There are already integrations that link HAProxy's metrics with Grafana or Kibana, so you won't need to build your own. You only need to set up your chosen software, install the plugin, and start using your data. Let's go over the steps for both options.

Grafana

Grafana is a popular choice for building graphs and dashboards and it supports Prometheus as a data source. So, you can leverage HAProxy's Prometheus feature. It works like this: HAProxy publishes its metrics at a known URL, **/metrics**. A Prometheus server scrapes this page every five seconds and stores the metrics over the long term, which allows you to calculate historical trends. Grafana fetches the data from the Prometheus server to display graphs.

In this tutorial, we will use the prebaked HAProxy dashboard from Ricardo F.'s *grafana-dashboards* code repository.

- Follow the Prometheus installation instructions to set up the Prometheus server that will store your metrics.
- Check that the Prometheus exporter has been compiled into your version of HAProxy. If not, you'll need to compile it in. Check by passing the -vv flag to HAProxy:

```
$ haproxy -vv | grep "Prometheus exporter"

Built with the Prometheus exporter as a service
```

- Add a *frontend* to your HAProxy configuration that listens on port 8404. This serves two purposes. It enables the HAProxy Stats page and also the Prometheus metrics web page. Configure it as shown below:

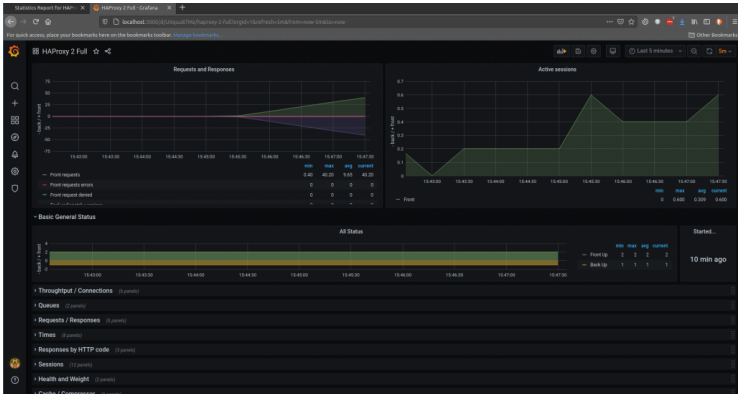
```
frontend stats
  bind :8404
  stats enable
  stats uri /
  stats refresh 5s
  http-request use-service prometheus-exporter
↪ if { path /metrics }
```

- On your Prometheus server, edit the file **/etc/prometheus/prometheus.yml** so that it includes your HAProxy server in the list of *targets* that it scrapes. Then restart the service. Here's how the file should look:

```
global:
  scrape_interval:     5s
  evaluation_interval: 5s

scrape_configs:
  - job_name: 'haproxy'
    static_configs:
      - targets: ['172.25.0.11:8404']
```

- Follow the Grafana installation instructions to set up a Grafana server. Once up and running, log in at port 3000 using the username and password *admin / admin*.
- Under the *Configuration* tab, add a new Prometheus Data Source. Set the URL to your Prometheus server. Then save it.
- Download Ricardo F.'s HAProxy Dashboard JSON file from GitHub, which is a prebaked dashboard with graphs for the HAProxy metrics.
- Go to the *Dashboard* > *Import* screen and paste the JSON into the textbox. Then click *Load*.

The imported dashboard displays many graphs, including those that show the metrics that we discussed earlier as being important for monitoring APIs.



Kibana

Kibana is the dashboard component of the Elastic Stack, a popular suite of tools for indexing and examining logs and metrics. To use it, we'll push HAProxy's metrics to an Elasticsearch database and then have Kibana display them.

We will use Metricbeat to ship HAProxy's metrics to Elasticsearch. Metricbeat's HAProxy module uses the Runtime API behind the scenes. Here's how to set it up:

- Enable the HAProxy Runtime API by adding the *stats socket* directive to the *global* section of your configuration. In the example below, it listens on port 9999:

```
global
  stats socket ipv4@*:9999 user haproxy group
↪ haproxy mode 660 level admin
```

- Deploy an Elasticsearch server. This is the database that will hold your metrics data over the long term.
- Deploy Kibana on a server.
- On your HAProxy server, install Metricbeat.
- Enable the HAProxy Metricbeat module with the following command:

```
$ metricbeat modules enable haproxy
```

- Edit the file **/etc/metricbeat/metricbeat.yml** so that it lists your Elasticsearch server under the *output.elasticsearch* section:

```
output.elasticsearch:
  hosts: ["172.25.0.19:9200"]
```

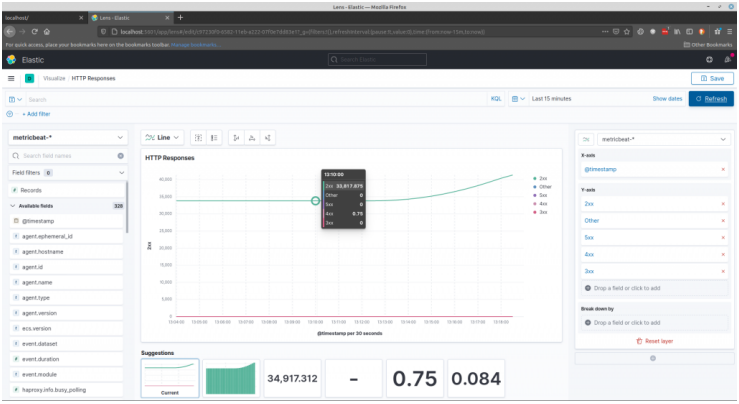
- Edit the file **/etc/metricbeat/modules.d/haproxy.yml** to configure the HAProxy module so that the *hosts* field includes the address and port where your HAProxy Runtime API is listening:

```
- module: haproxy
  metricsets: ["info", "stat"]
  period: 10s
  hosts: ["tcp://172.25.0.11:9999"]
  enabled: true
```

- Start the Metricbeat service to begin shipping HAProxy metrics to Elasticsearch:

```
$ sudo service metricbeat start
```

- Open Kabana in your browser. It listens at port 5601. Go to *Kibana* > *Dashboard* and click to Create index pattern. Set the index pattern to *metricbeat-**.
- On the next screen, set the Time field to *@timestamp*. You should now see HAProxy metrics when you search for them.
- Go to *Kibana* > *Visualize* and create a new visualization. Try the Lens dashboard, which offers an intuitive way to create graphs from your metrics. You can search for metrics that begin with *haproxy* and then drag them onto the visualization canvas.
- In the screenshot below, I have dragged the *haproxy.stat.response.http.2xx*, *3xx*, *4xx*, and *5xx* metrics onto the visualization.



Conclusion

HAProxy provides many metrics that are essential when proxying API traffic. A few that you should consider are server health status, request rate, number of errors, average response time, and number of retries. This data can be extracted from HAProxy either through its Runtime API or the built-in Prometheus exporter. There are already integrations available for popular, open-source graphing tools such as Grafana and Kibana. With minimal setup, you can begin observing trends in your data.

Caching

Something else that HAProxy adds is the ability to cache API responses, which can boost how quickly clients receive data. In this chapter, you will learn how to set up HAProxy's cache feature, which speeds up delivery of messages and lessens the load placed upon your backend servers.

Why you should cache API responses

When it comes to web applications, there are two readily available caches: a client-side (browser) cache and a server-side (HAProxy) cache. A browser's cache will boost performance for a single user. HAProxy's cache, which is known as a *proxy cache*, will speed it up for all users because once a resource is cached in the proxy, it's available for anyone making the same request until it expires. It's easy to enable, but you should know how to use it effectively.

HAProxy's cache runs in memory, which makes it fast. Other proxy caches need to read and write state on the filesystem, which incurs some I/O latency. Also, because it runs within HAProxy, you don't need to contact an upstream cache server, which means you have one less transfer across the network. In some cases, however, you'll want the extra features of a shared cache server like

Varnish. HAProxy's caching feature is modest in comparison, but it might be exactly what you need for caching API responses.

Why should you cache API responses?

For one thing, it will speed up the time components take to receive their data, which has a huge effect on how responsive your website seems overall. One of the biggest obstacles to adopting a component-based design (i.e. Vue, React, Angular) is the fear that your webpage will render its initial HTML page quickly, but linger in an unusable state while the individual components wait to load. A lot of that time waiting is spent processing the request on the web server, pulling the requested data out of the database, and forming the JSON-encoded response. Caching allows you to perform those steps only once and then serve the saved message to other clients.

Another reason to love proxy caching is because it reduces load on your servers. They don't need to process nearly as many requests, many of which are likely the same request they saw earlier. It's perfectly fine to serve a slightly stale response for content that doesn't change extremely often, such as daily news feeds, product descriptions, reviews, and comment boards. Even caching this content for five or ten seconds could have a worthwhile impact, depending on the number of users viewing that same data. By the way, caching for a very short period of time is known as *microcaching*.

API functions that return data, rather than modify it, are best suited for caching. This typically includes any function

called with GET. Just be sure that your API responses do not include any information that is specific to a user, such as API keys, user profile data, and the like.

How to cache with HAProxy

In your HAProxy configuration file, add a cache section. It goes at the same level as a global or defaults section. You can have more than one cache section to create multiple caches for different purposes, and each can set its own max-age and other attributes.

```
global
  # global settings

defaults
  # default settings

cache mycache
  total-max-size 4095   # MB
  max-object-size 10000 # bytes
  max-age 30            # seconds
```

The *total-max-size* directive sets the total amount of memory that this cache can consume; It has a maximum value of 4095 megabytes. The *max-object-size* directive sets the largest size of a single item you can store in the cache, and it can only be half of the *total-max-size* value. In this example, I've set it to 10,000 bytes, which is 10 kilobytes. If a response is larger, it simply won't be cached. The last directive, *max-age*, sets the time-to-live (TTL) in seconds for

an item in the cache. After the TTL expires, the item will be removed from memory.

Next, add an *http-request cache-use* and an *http-response cache-store* directive to your backend section. The former uses a cached resource if it's found and the latter adds it to the cache. Both take the name of a cache section.

```
frontend fe_api
  bind :80
  default_backend be_api

backend be_api
  # Get from cache / put in cache
  http-request cache-use mycache
  http-response cache-store mycache

  # server list
  server s1 172.25.0.10:8080 check
```

You can also restrict which responses should be cached by appending an *if* statement to the end of the *http-request cache-use* directive. For instance, if you wanted to cache only when the requested URL path begins with **/api/news_feed/**, you would use the following:

```
http-request cache-use api_cache
↪ if { path_beg /api/news_feed/ }
http-response cache-store api_cache
```

Notice that you add a condition to the *http-request* line, but you do not need one on the *http-response* line. HAProxy is designed to skip caching if there's no chance the item will ever be used. Alternatively, the backend server can return a Cache-Control header with a *no-store* attribute to disable caching of a particular response.

Cache-Control: no-store

The Cache-Control header also supports the *s-maxage* attribute, which lets you override the TTL that was set in HAProxy's cache section. Consider the following Cache-Control header, which allows the response to be cached, but sets its TTL to 10 seconds:

Cache-Control: public,s-maxage=10

To see the TTL that was set on an item in the cache, call the HAProxy Runtime API *show cache* command, which shows the TTL as the *expire* field:

```
$ echo "show cache" |\
   socat tcp-connect:127.0.0.1:9999 -

0x7fcad7c9603a: api_cache (shctx:0x7fcad7c96000,
available blocks:4193280)
0x7fcad7c960c0 hash:3075548050 size:363 (1
blocks), refcount:0, expire:10
```

You can also get metrics about your cache, which can be displayed in Grafana. If you've enabled Prometheus metrics in HAProxy, scrape the following metrics from HAProxy's Prometheus endpoint (where the *proxy* label would be the name of your frontend or backend):

- haproxy_frontend_http_cache_lookups_total{proxy="fe_api"}
- haproxy_frontend_http_cache_hits_total{proxy="fe_api"}
- haproxy_backend_http_cache_lookups_total{proxy="be_api"}
- haproxy_backend_http_cache_hits_total{proxy="be_api"}

These metrics show you how many cache lookups were performed and how many resulted in a cache hit. You can use that to adjust your TTL values.

One last trick: You can return a response header that shows whether the requested resource was found in the cache. Add these two lines to your frontend:

```
http-response set-header X-Cache-Status HIT
↪ if { res.cache_hit -m bool }
http-response set-header X-Cache-Status MISS
↪ if !{ res.cache_hit -m bool }
```

HAProxy will set the *X-Cache-Status* header to *HIT* if the item was found in the cache, or to *MISS* otherwise.

Conclusion

HAProxy's cache helps boost the speed of your API services, resulting in a more responsive website.

Define how long responses should be cached using the *max-age* directive, which you can override with a *Cache-Control* header. If there are certain responses that should not be cached at all, you can use an *if* statement to filter them out or you can set your *Cache-Control* header to *no-store*.

The HAProxy Runtime API will show you how long items will live in the cache and HAProxy's Prometheus metrics endpoint exposes counters for lookups and cache hits. Now go and enjoy the benefits of proxy caching!

Authentication

APIs provide direct access to backend systems and may return sensitive information such as healthcare, financial and PII data. APIs often expose *create*, *update* and *delete* operations on your data too, which shouldn't be open to just anyone.

In this post, we'll demonstrate how HAProxy defends your APIs from unauthorized access with OAuth 2 access tokens and shrinks the attack surface that you might otherwise expose.

Authentication and authorization

Let's begin with a scenario where you have an API to protect. For example, let's say that this API provides methods related to listing pets up for adoption. It has the following API endpoints:

API endpoint	**What it does**
GET /api/pets	Returns a list of pets ready to be adopted
POST /api/pets/{name}	Adds a newly arrived pet to the list
DELETE	Removes a pet from the list after it

/api/pet/{name}	has found a home

This fictitious API lets you view available pets, add new ones to the list, and remove them after they've been adopted to loving homes. For example, you could call *GET /api/pets* like this:

GET https://api.example.com/api/pets

```
[
  { "species": "hamster", name: "lloyd" },
  { "species": "fish", name: "franklin" },
  { "species": "cat", name: "lisa" },
  { "species": "dog", name: "barney" },
]
```

This would be consumed by your frontend application, perhaps through Ajax or when loading the page. For requests like this that retrieve non-sensitive information, you may not ask users to log in and there may not be any authentication necessary.

For other requests, such as those that call the POST and DELETE endpoints for adding or deleting records, you may want users to log in first. If an unauthenticated user tries to call the POST and DELETE API methods, they should receive a *403 Forbidden* response.

There are two terms that we need to explain: authentication and authorization. *Authentication* is the process of getting a user's identity. Its primary question is: *Who* is using your API? *Authorization* is the process of granting access. Its primary question is: Is this client approved to call your API?

OAuth 2 is a protocol that authenticates a client and then gives back an access token that tells you whether or not that client is authorized to call your API. For the most part, the concept of *identity* doesn't play a big part in OAuth 2, which is concerned with authorization. Think of it like going to the airport, and at the first gate you are meticulously inspected by a number of set criteria. Upon inspection, you are free to continue on to your terminal where you can buy overpriced coffee, duty-free souvenir keychains and maybe a breakfast bagel. Since you've been inspected and have raised no red flags, you are free to roam around.

In a similar way, OAuth 2 issues tokens that typically don't tell you the identity of the person accessing the API. They simply show that the user, or the client application that the user has delegated their permissions to, should be allowed to use the API. That's not to say that people never layer on identity properties into an OAuth token. However, OAuth 2 isn't officially meant for that. Instead, you would use something like OpenID Connect, which is layered on top of OAuth 2, if you need identity information.

As we described in earlier chapters, an *API gateway* is a proxy between the client and your backend API services that routes requests intelligently. It also acts as a security

layer. When you use HAProxy as your API gateway, you can validate OAuth 2 access tokens that are attached to requests.

For simplifying your API gateway and keeping the complicated authentication pieces out of it, you'll offload the task of authenticating clients to a third-party service like Auth0 or Okta. These services handle logging users in and can distribute tokens to clients that successfully authenticate. A client application would then include the token with any requests it sends to your API.

Let's see how to configure HAProxy to validate access tokens.

OAuth 2 access tokens

An OAuth 2 access token uses the JSON Web Token (JWT) format and contains three base64-encoded sections:

- A *header* that contains the type of token (JWT in this case) and the algorithm used to sign the token;
- A *payload* that contains:
    - the URL of the token issuer
    - the audience that the token is intended for (your API URL)
    - an expiration date
    - any scopes (e.g. read and write) that the client application should have access to;

- A *signature* to ensure that the token is truly from the issuer and that it has not been tampered with since being issued.

We won't focus on how a client application gets a token. In short, you'd redirect users to a login page hosted by a third-party service like Auth0 or Okta. Instead, we'll highlight how to validate a token. You will see how HAProxy can inspect a token that's presented to it and then decide whether to let the request proceed.

An access token contains many fields, but a few that are most interesting are:

- *alg*, the algorithm, which is often set to *RS256*, that was used to sign the token;
- *iss*, the issuer, or the service that authenticated the client and created the token;
- *aud*, the audience, which is the URL of your API gateway;
- *exp*, the expiration date, which is a UNIX timestamp;
- *scope*, which lists the granular permissions that the client has been granted (Note that Okta calls this field "scp", so the Lua code that we'll use would have to be modified to suit).

API gateway sample application

To follow this tutorial, you have two options:

- You can clone the sample application from Github and use *Vagrant* to set it up.
- You can clone the JWT Lua code repository by itself. It provides an install script to assist with installing the Lua library and its dependencies into your own environment.

The workflow for authorizing users looks like this:

1. A client application uses one of the Client Credentials Flow to request a token from the authentication service.
2. Once the client has received a token, it stores it so that it can continue to use it until it expires.
3. When calling an API method, the application attaches the token to the request in an HTTP header called *Authorization*. The header's value is prefixed with *Bearer*, like so:

   Authorization: Bearer <token>

4. HAProxy receives the request and performs the following checks:

    a. Was the token signed using an algorithm that the Lua code understands?
    b. Is the signature valid?
    c. Is the token expired?
    d. Is the issuer of the token (the authenticating service) who you expect?
    e. Is the audience (the URL of your API gateway) what you expect?
    f. Are there any scopes that would limit which resources the client can access?
5. The application continues to send the token with its requests until the token expires, at which time it repeats Step 1 to get a new one.

To test it out, sign up for an account with Auth0. Then, you can use *curl* to craft an HTTP request to get a new token using the *client credential* grant flow. POST a request to **https://{your_account}.auth0.com/oauth/token** and get an access token back. The Auth0 website gives you some helpful guidance on how to do this.

Here's an example that asks for a new token via the **/oauth/token** endpoint. It sends a JSON object containing the client's credentials, client_id and client_secret:

```
$ curl --request POST \
   --url 'https://myaccount.auth0.com/oauth/token'
\
   --header 'content-type: application/json' \
   --data '{"client_id": "abcdefg12345",
↪ "client_secret": "HIJKLMNO67890", "audience":
↪ "https://api.example.com", "grant_type":
↪ "client_credentials", "scope": "read:pets
↪ write:pets"}'
```

You'll get back a response that contains the JWT access token:

```
{
  "access_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJS...",
  "scope": "write:hamsters read:hamsters",
  "expires_in": 86400,
  "token_type": "Bearer"
}
```

In a production environment, you would use the Client Credentials workflow only with trusted client applications where you can protect the client ID and secret.

Now that you have a token, you can call methods on your API. One of the benefits of OAuth 2 over other authorization schemes like session cookies is that you control the process of attaching the token to the request.

Whereas cookies are always passed to the server with every request, even those submitted from an attacker's website as in CSRF attacks, your client-side code controls sending the access token. An attacker will not be able to send a request to your API URL with the token attached.

A POST request that includes the token will look like this:

```
curl --request POST \
  --url https://api.example.com/api/pets \
  --data '{ "species": "fish", name: "wanda" }' \
  --header 'authorization: Bearer
↪ eyJ0eXAiOiJKV1QiLCJhbGciOiJS...'
```

In the next section, you'll see how HAProxy can, with the addition of some Lua code, decode and validate access tokens.

Configure HAProxy for OAuth 2

Before an issuer like Auth0 gives a client an access token, it signs it. Since you'll want to verify that signature, you'll need to download the public key certificate from the token issuer's website. On the Auth0 site, you'll find the download link under **Applications > [Your application] > Settings > Show Advanced Settings > Certificates**. Note, however, that it will give you a certificate in the following format:

```
-----BEGIN CERTIFICATE-----
MIIDATCCAemgAwIBAgIJOTQvWZNFMdgBMA0GCSqGSIb3DQEBCw
UAMB4xHDAaBgNVBAMTE25pY2tyYW00NC5hdXRoMC5jb20wHhcN
MTgxMDA5MDA1OTMyWhcNMzIwNjE3MDA1OTMyWjAeMRwwGgYDVQ
QDExNuaWNrcmFtNDQuYXV0aDAuY29tMIIBIjANBgkqhkiG9w0B
AQEFAAOCAQ8AMIIBCgKCAQEAvIL8bebCh+pi68Rt0CCu104VqR
10kuD0
E1yzwaywvaEiyhfUeDDKAyKC8yS5ilu9xyWK/pg/84RiWq7Woq
hUm8L06jtknn/ZCOuyUdkn1QcdOG10lbbrUF1AOduTIvFYyT4z
HrIcKt6MyeQUO0kHcXQU7lvM2C62BboAasZFupDts1m1kPZMWa
iSjLrE1eruhl8NrfipiPWMZJSJoYCQcmtN3REXk9z8X7ZPgcMJ
9hNN+Kv0fTYLZI4wS4TpHscVfbK18cL4uLrTCcip7jNey2KZ/Y
dbeHgmmcQAdiB4veH4I2dAyqIdsy8Jk+KTs3Ae8qp+S3XtC8z/
uXMbN7lRAwIDAQABo0IwQDAPBgNVHRMBAf8EBTADAQH/MB0GA1
UdDgQWBBRh4OxTHcFgxEk96rKbvWHibUeBwzAOBgNVHQ8BAf8E
BAMCAoQwDQYJKoZIhvcNAQELBQADggEBACYMzTV0kHcRDwJyj+
XHmmFimPCcgOPOwo4h4eSRIq8XCyFhdOlhuyj8T6ESClKaAz5O
mKvXBBP7OnpkUcrbv1VaNCluc/X6in2hptru3L/Ouxjv22QwCW
NVB288ns3cYszr5M1ycaWnqXDmY4/xoK3phUcTIQBFY1I1JuKx
DzSihDeEAlkXMYwiCSreG1WuAmyA3oWEfdpfnwwz3QT2YTRs3P
/IKSlLeYzC1Wn5BYrmyHK1EC7scTofdFz+OqldINLB08kk7Axv
73hwD72zNfYVzX9Eh+d3jH6u6TsLD2M6dvTvYyMP8yRLy1LbbR
paZBfFdDrEtqOO0+61o9gGYJE=
-----END CERTIFICATE----
```

This contains the public key that you can use to validate the signature, but also extra metadata that can't be used. Invoke the following OpenSSL command to convert it to a file containing just the public key:

```
$ openssl x509 -pubkey -noout \
   -in ./mycert.pem > pubkey.pem
```

This will give you a new file called **pubkey.pem** that is much shorter:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvIL8be
bCh+pi68Rt0CCu104VqR10kuD0E1yzwaywvaEiyhfUeDDKAyKC
8yS5ilu9xyWK/pg/84RiWq7WoqhUm8L06jtknn/ZCOuyUdkn1Q
cdOG10lbbrUF1AOduTIvFYyT4zHrIcKt6MyeQUO0kHcXQU7lvM
2C62BboAasZFupDts1m1kPZMWaiSjLrE1eruhl8NrfipiPWMZJ
SJoYCQcmtN3REXk9z8X7ZPgcMJ9hNN+Kv0fTYLZI4wS4TpHscV
fbK18cL4uLrTCcip7jNey2KZ/YdbeHgmmcQAdiB4veH4I2dAyq
Idsy8Jk+KTs3Ae8qp+S3XtC8z/uXMbN7lRAwIDAQAB
-----END PUBLIC KEY-----
```

In the sample project, I store this file in the **pem** folder and then Vagrant syncs that folder to the VM. I then use an environment variable to tell the Lua code where to find it. In fact, I use environment variables for passing in several other parameters as well. Use *setenv* in your HAProxy configuration file to set an environment variable:

```
global
  lua-load /usr/local/share/lua/5.3/jwtverify.lua
  setenv OAUTH_PUBKEY_PATH
↪ /usr/local/etc/haproxy/pem/pubkey.pem
  setenv OAUTH_ISSUER https://myaccount.auth0.com/
  setenv OAUTH_AUDIENCE https://api.mywebsite.com
```

A *lua-load* directive loads a Lua library called **jwtverify.lua** that contains code for validating access tokens. Get this library from the JWT Lua code repository.

Next, the frontend receives requests on port 443 and performs various checks by invoking the **jwtverify.lua** file. Here we're using ACL statements to define conditional logic that allows or denies a request. ACLs are a powerful and flexible system within HAProxy and one of the building blocks that make it so versatile.

```
frontend api_gateway
  # Use HTTPS to protect the secrecy of the token
  bind :443 ssl crt
↪ /usr/local/etc/haproxy/pem/test.com.pem

  # Accept GET requests and skip further checks
  http-request allow if { method GET }

  # Deny the request if it's missing an
  # Authorization header
  http-request deny unless {
req.hdr(authorization)
↪ -m found }

  # Verify the token
  http-request lua.jwtverify
  http-request deny unless { var(txn.authorized)
↪ -m bool }

  # (Optional) Deny the request if it's a
  # POST/DELETE to a path beginning with
  # /api/pets, but the token doesn't
  # include the "write:pets" scope
  http-request deny if { path_beg /api/pets } {
↪ method POST DELETE } ! { var(txn.oauth_scopes)
↪ -m sub write:pets }

  # If no problems, send to the apiservers backend
  default_backend apiservers
```

The first *http-request deny* line rejects the request if the client did not send an *Authorization* header at all. The next line, *http-request lua.jwtverify*, invokes the Lua library, which performs the following actions:

- decodes the JWT,
- checks that the algorithm used to sign the token is supported (RS256),
- verifies the signature,
- ensures that the token is not expired,
- compares the issuer in the token to the OAUTH_ISSUER environment variable,
- compares the audience in the token to the OAUTH_AUDIENCE environment variable,
- if any scopes are defined in the token, adds them to an HAProxy variable called *txn.oauth_scopes* so that subsequent ACLs can check them,
- if everything passes, sets a variable called *txn.authorized* to true.

The next *http-request deny* line rejects the request if the Lua library did not set a variable called *txn.authorized* to a value of *true*. Notice how booleans are evaluated by adding the *-m bool* flag.

The next two lines reject the request if the token does not contain a scope that matches what we expect for the HTTP path and method. Scopes in OAuth 2 allow you to define specific access restrictions. In this case, POST and DELETE requests require the *write:pets* permission. Scopes are

optional and some APIs don't use them. You can set them up on the Auth0 website and associate them with your API. If the client should have these scopes, they'll be included in the token.

To summarize, any request for **/api/pets** must meet the following rules:

- It must send an *Authorization* header that contains a JWT access token
- The token must be valid
- The token must contain a scope that matches what you expect

With this configuration in place, you can use *curl* to send requests to your API, attaching a valid token, and expect to get a successful response. Using this same setup, you'd lock down your APIs so that only authenticated clients can use them.

Conclusion

In this chapter, you learned more about using HAProxy as an API gateway, leveraging it to secure your API endpoints using OAuth 2.

Clients request tokens from an authentication server, which sends back a JWT. That token is then used to gain access to your APIs. With the help of some Lua code, HAProxy can validate the token and protect your APIs from unauthorized use.

Monetization

In the previous chapter, you learned that when you operate HAProxy as an API gateway, you can restrict access to your APIs to only clients that present a valid OAuth 2 access token. In this chapter, we take it a step further. You will learn how to leverage tokens to grant some users more access than others and then charge for the service. This is called *API monetization* and it's one way to turn your APIs, and the data that they expose, into a profitable enterprise.

Set up the demo project

You'll find the example code in our GitHub repository. We use Docker Compose to create the following components in a self-contained, virtual network:

- an HAProxy server, which acts as an API gateway,
- three API servers behind HAProxy,
- a Keycloak server, also behind HAProxy, which acts as our authentication server.

Here's how it all fits together: When HAProxy receives an API call—which is any HTTP request that has a URL beginning with **/api/**—it relays it to one of the three API servers behind it. However, clients must attach *access tokens*, which are like digital members-only cards, to their requests before HAProxy will grant them access. An access token bestows certain privileges to the client that presents it. In our demo, each client has a right to make HTTP

requests to our service, but some clients are allowed to make more requests per minute than others depending on their token.

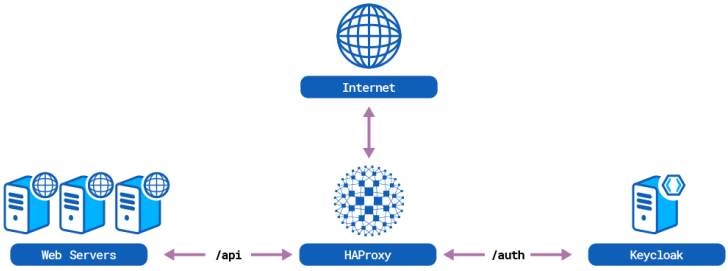
HAProxy can validate a token, read its properties, and make decisions based on those properties. We imprint our tokens with a subscription level: bronze, silver, or gold. Bronze is the lowest tier and clients who present a token with that level are granted only 10 requests per minute. Silver clients get 100 requests per minute and those with a gold level are allowed 1000 requests per minute.

That's one end of the equation. The other is the party that creates the tokens. As you saw in the previous chapter about authentication and authorization, HAProxy will work with a variety of OAuth 2 token providers, including Auth0 and Okta. In this post, we use a self-hosted authentication server called Keycloak. We place our Keycloak server behind HAProxy and whenever a client requests a URL beginning with **/auth/**, HAProxy routes it there. Typically, these requests are either a client requesting a token or you, the administrator, adding clients to the system.

With these subscription levels in place and HAProxy granting access only to clients that have a token, you can of course start charging a fee to access your service. *Et voilà*! API monetization. Let's see how to set it up!

Before a client can send a request to your API servers, they must authenticate with Keycloak, get an OAuth 2 access token from it, and present that token to HAProxy. HAProxy

then verifies whether the token is valid before allowing the client to proceed with their request.



First, download the sample project and then initialize the components by calling Docker Compose:

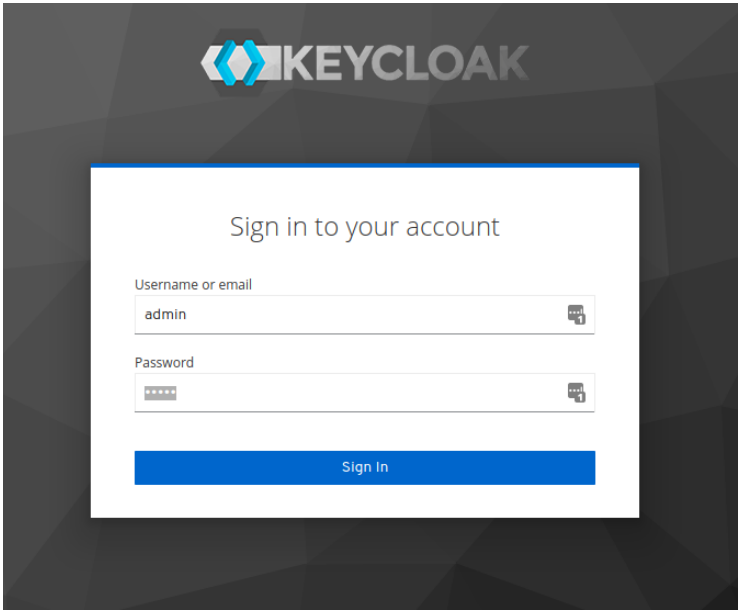
```
$ git clone
↪ https://github.com/haproxytechblog/haproxy-api-
↪ monetization-demo.git
$ cd haproxy-api-monetization-demo
$ sudo docker-compose up -d
```

These commands start up HAProxy, Keycloak, and the API servers.

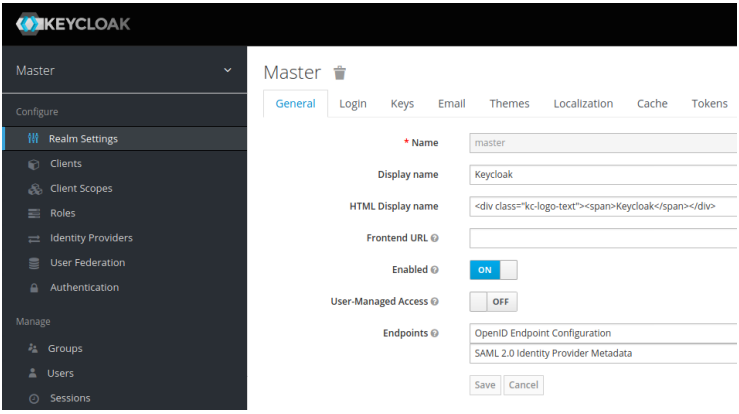
Configure Keycloak

Once the demo is up and running, go to **http://localhost/auth/** and log into the Keycloak

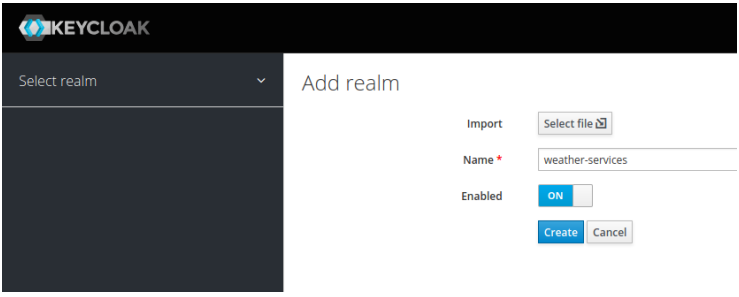
Administration Console with the username and password *admin*.



When you first log in, you'll see the configuration screen for the top-most *realm*. From here, you have full access to all of Keycloak.



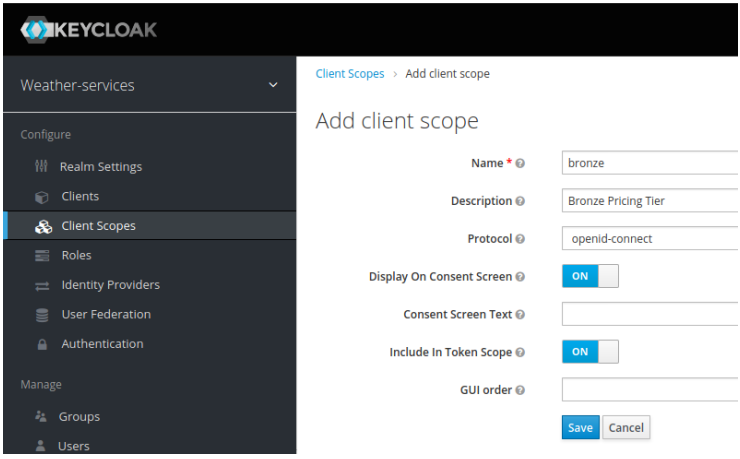
Next, you'll need to create new realms for each service that you want to monetize. Within each realm, you can add authorized clients. First, click the top-level dropdown menu where it says *Master* and choose *Add realm*. For this example, set the new realm's name to *weather-services*. Then click *Create*.



You are taken to the *Realm Settings* page for the *weather-services* realm.

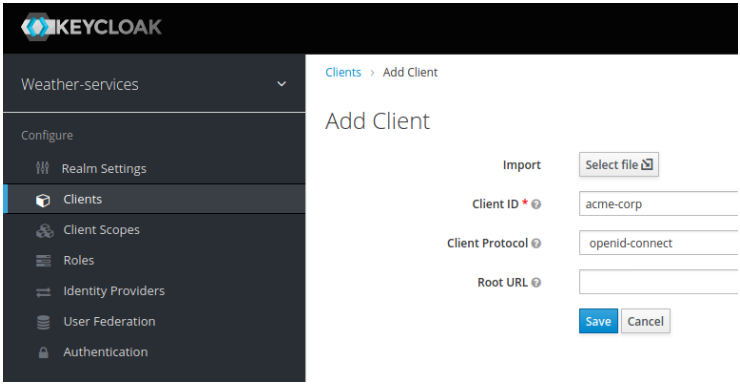
From here click the *Tokens* tab, set the *Default Signature Algorithms* field to *RS256*, and then click *Save*. Keycloak will now sign its access tokens with its private key and, later, HAProxy will use Keycloak's public key to verify that signature.

Click the *Client Scopes* link next. Add three scopes—bronze, silver, and gold—which will serve as different pricing tiers for accessing your Weather Services APIs. Click the *Create* button and add each of these scopes.



After you've created the bronze, silver, and gold scopes, click the *Clients* link. In the most technical sense, a client is an application that accesses your services. For example, the client might be a web application that uses your API to get up-to-date weather forecasts. More broadly, a client may be a customer who has signed up to call your service, and they may do so from multiple applications.

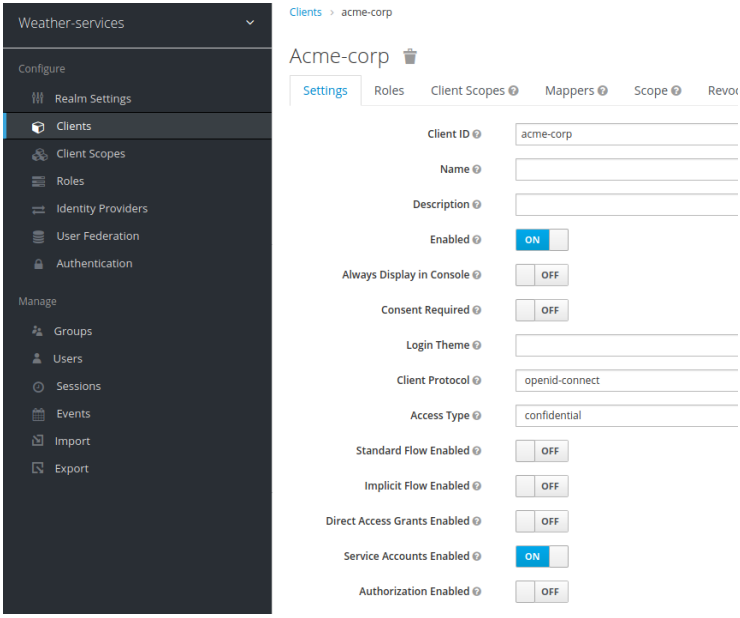
Click the *Create* button on the *Clients* screen to add a new client.



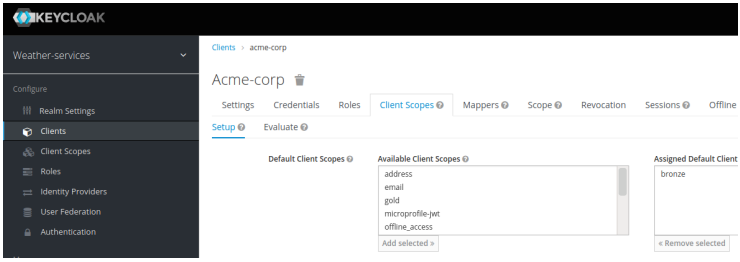
When adding a client, you're asked to assign a unique Client ID. This can be any string, such as the organization's name, an email address, or a GUID. The client will use this ID when they access your services, so it must be something you don't mind sharing with them. In this exercise, I set the Client ID to *acme-corp*.

After you've created the client, you're taken to the client's *Settings* screen. Because we want to enable machine-to-machine authentication, you must enable the OAuth 2 Client Credentials grant. That's what OAuth calls the workflow for allowing an application to request an access token. To enable this on the *Settings* screen, change the *Access Type* field to *confidential* and set *Service Accounts Enabled* to *on*.

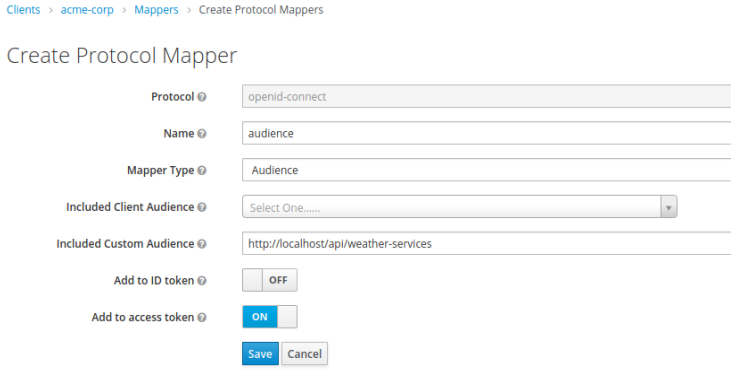
You can set *Standard Flow Enabled* and *Direct Access Grants Enabled* to off. We won't be using those types of grants.



Click *Save* at the bottom of the screen and then on the *Client Scopes* tab, add the *bronze* scope. Remove all of the other previously assigned client scopes. One peculiar behavior of Keycloak is that by default it assigns the *roles* client scope, which has the effect of adding a second value to the *aud* field in the token, which you don't want. Play it safe and remove all extra scopes.



Next, go to the *Mappers* tab and create a new mapper. Set its *Mapper Type* field to *Audience* and its *Included Custom Audience* field to the URI of your API service. In this example, I set it to **http://localhost/api/weather-services**. The audience value will need to match what we hardcode in the HAProxy configuration. It's one way that HAProxy validates the token.



Now you're ready to play the role of a client requesting access to your services. We've given the *acme-corp* client a scope called *bronze*, which will mean they're allowed up to

10 API calls per minute. That rate limit is handled by HAProxy.

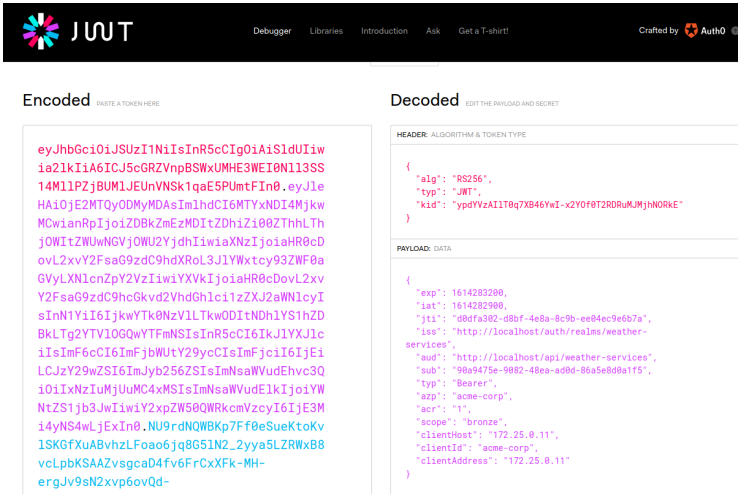
Get an access token

Now that you've configured a client in Keycloak, you can try out getting an access token. First, copy the *Client ID* from the *acme-corp* Client page and the *Secret* from the Credentials page. Then, use curl to request a new access token, providing the *client_id* and *client_secret* fields with your request. Note that we set the *grant_type* field to *client_credentials*:

```
$ curl --request POST \
   --url
↪ 'http://localhost/auth/realms/weather-services/
↪ protocol/openid-connect/token' \
   --data 'client_id=acme-corp' \
   --data 'client_secret=7f2587ee-a178-4152-bd91-
↪ 7b758c807759' \
   --data 'grant_type=client_credentials'

{"access_token":"eyJhbGciOiJSUzI1NiIsI...","expire
s_in":300,"refresh_expires_in":0,"token_type":"Bea
rer","not-before-policy":0,"scope":"bronze"}
```

The request returns a JSON document that includes an access token. The token is encoded, but you can decode it

by pasting it into the *Encoded* textbox on the https://jwt.io/ website. An example of the decoded fields is shown below.



You'll find three of the fields especially interesting:

- *iss* is the issuer, or the service that authenticated the client and created the token; In this case, it's set to **http://localhost/auth/realms/weather-services**, which is the Keycloak realm for our *weather-services* API.
- *aud* is the audience, which is the URL of your API gateway; In this case, it's set to **http://localhost/api/weather-services**.
- *scope* is the list of permissions granted to the client; It includes *bronze*.

The *scope* field includes the *bronze* client scope, which we will use when setting a rate limit for this client.

Configure access in HAProxy

First, you need to configure HAProxy so that it limits access to your services to only authorized clients. Install the JSON Web Token (JWT) library into HAProxy, which is a Lua library that inspects incoming OAuth 2 access tokens that are attached to HTTP requests. The library's GitHub page has instructions for how to install it, but in the demo project the library is already installed as part of the Docker container's image.

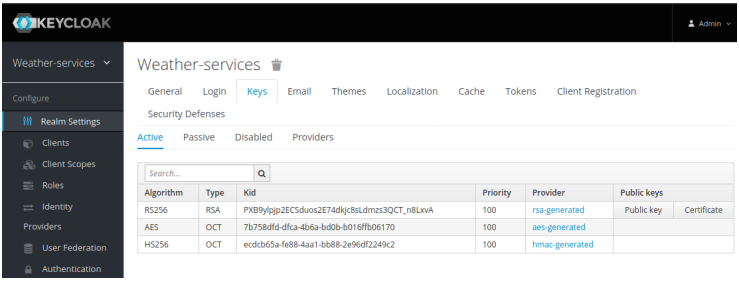
Next, configure the library. In the global section of your HAProxy configuration file, use the setenv directive to define the issuer, audience and public key that HAProxy should use when validating tokens.

```
global
  lua-load /usr/local/share/lua/5.3/jwtverify.lua

  setenv OAUTH_ISSUER
↪ http://localhost/auth/realms/weather-services
  setenv OAUTH_AUDIENCE
↪ http://localhost/api/weather-services
  setenv OAUTH_PUBKEY_PATH
↪ /etc/haproxy/pem/pubkey.pem
```

The issuer and audience must match the token's *iss* and *aud* fields exactly or else the token won't be accepted. That ensures that the token comes from a trusted souce (Keycloak) and is meant for our API only.

HAProxy uses the public key to verify the digital signature on the token. Keycloak uses its private key to sign the access tokens it gives to clients. HAProxy verifies that signature by comparing it with Keycloak's public key, which it stores locally.

In our example project, the public key, **pubkey.pem**, is mounted as a volume into the HAProxy container. Download the key from Keycloak by going to the *weather-services* **Realm Settings > Keys** page and clicking the *Public key* link on the row that says RS256.



Replace the contents of the file **pubkey.pem** in the demo project with the value from Keycloak. You must prefix the value with *—–BEGIN PUBLIC KEY—–* and end it with *—–END PUBLIC KEY—–*, as shown here:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoG0pxf
K54qjF8gUzjARji3D2VZ9x7UTRE+75SoIcSHkPWg8Dlb/DzDNp
ofG8bB3FyvcqihF0sFTnbQG4+2XKODuxeG2o609YhGvai0hHNZ
FXZEANMAoRSEdSq3oCDiAladKez92VjxDjo3W9zLvFhAAYEGQD
BvRTqbbHhsCm5fm2k7A3wMB5H0G/i2x6ZDD5tA7LsIngyJwELS
IjFzIfP8xylJdppWwQFJEjYagCXahO4WW+oOMpFs+X1gJ3xBlN
6pLsVSNWrKZMe/qpZDzQO8qnGoVI7tkZpCkR62B7OyVGzjDB0N
wJTwN787xuTURsDNF0Gm3rFgSVnVokn07rqQIDAQAB
-----END PUBLIC KEY-----
```

Next, in the frontend section where you want to restrict client access, add the following configuration directives:

```
frontend fe_api
  bind :80

  # a stick table stores the count of requests
  # each clients makes
  stick-table  type ipv6  size 100k  expire 1m
↪ store http_req_cnt

  # allow 'auth' request to go straight through
  # to Keycloak
  http-request allow if { path_beg /auth/ }
  use_backend keycloak if { path_beg /auth/ }

  # deny requests that don't send an access token
  http-request deny deny_status 401 unless {
↪ req.hdr(authorization) -m found }

  # verify access tokens
  http-request lua.jwtverify
  http-request deny deny_status 403 unless {
↪ var(txn.authorized) -m bool }

  # add the client's subscription level to
  # the access logs: bronze, silver, gold
  http-request capture var(txn.oauth_scopes) len
10

  # deny requests after the client exceeds
  # their allowed requests per minute
  http-request deny deny_status 429 if {
↪ var(txn.oauth_scopes) -m sub bronze } {
```

```
↪ src,table_http_req_cnt gt 10 }

  http-request deny deny_status 429 if {
↪ var(txn.oauth_scopes) -m sub silver } {
↪ src,table_http_req_cnt gt 100 }

  http-request deny deny_status 429 if {
↪ var(txn.oauth_scopes) -m sub gold }   {
↪ src,table_http_req_cnt gt 1000 }

  # track clients' request counts. This line
  # will not be called once the client is
  # denied above, which prevents them from
  # perpetually locking themselves out.
  http-request track-sc0 src

  default_backend be_api
```

This configuration requires that all requests include a valid, non-expired access token. It also checks the token's scopes to see which subscription level was assigned. Bronze level allows up to 10 requests per minute, silver allows 100 requests per minute, and gold allows 1000 requests per minute.

Restart the HAProxy Docker container to load the new settings:

```
$ sudo docker-compose restart haproxy
```

Make a request

Try it out by first getting an access token using the following curl command:

```
$ curl --request POST \
   --url 'http://localhost/auth/realms/weather-
↪ services/protocol/openid-connect/token' \
   --data 'client_id=acme-corp' \
   --data 'client_secret=9e9e2acc-cd15-4878-9e5a-
↪ c815d29a976f' \
   --data 'grant_type=client_credentials'
```

Copy the access token from the response and paste it into the following command where it says [ACCESS_TOKEN]:

```
$ curl --request GET \
   --url http://localhost/api/weather-services
↪ /43213 \
   --header 'authorization: Bearer [ACCESS_TOKEN]'
```

You should get back a valid JSON response. If not, check HAProxy's logs with the *docker-compose logs haproxy* command. Since we configured the *acme-corp* client to have *bronze* access, you can make only 10 requests per

minute, after which you will get a *429 Too Many Requests* error.

Try assigning the silver or gold scope to the acme-corp client via the Keycloak Administration Console, fetching a new token, and then retrying the GET request. You should be allowed more requests per minute.

Conclusion

When you use HAProxy as an API gateway, you can validate OAuth 2 access tokens. By imprinting subscription levels into the tokens, you can monetize your APIs, charging a fee for expanded access. Monetization can be a smart move once your APIs reach a certain level of popularity, and you can even continue to offer a free tier to entice newcomers. With HAProxy, you can layer on this functionality at any point.

API monetization can take many forms and rate limiting is only one aspect. Yet, it's one that's used successfully by many companies. To protect your customers, you'll want to add security protections that deter bots and malicious users.

Security

In almost every case, APIs have changed how modern applications connect to their data. Mobile apps, single-page web apps, IoT devices, integration hooks between software—all of these things rely on APIs to fetch, update, delete, and create data. In fact, one set of APIs might serve as the backbone of a website, mobile app, voice assistant device, and more, meaning one data store owns a treasure trove of information about us, the human users.

All of this to say that APIs are irresistible targets to would-be attackers. They would love to gain access to that data: to steal it, to categorize it, and to sell it.

In this chapter, we'll discuss ways to improve the security of your APIs. We'll see how placing your servers behind HAProxy and using it as an API gateway lets you narrow the point of entry for attackers. From this defensive position, you can build up your countermeasures. We will discuss the benefits of enabling authentication, TLS encryption, rate limiting, anomalous behavior protection, a web application firewall, and bot management in your API gateway.

Authentication

Who should have access to your APIs?

Few APIs allow open access to anonymous users. Even free APIs that publish non-sensitive data will benefit from requiring some form of login. That's because at the very least it gives you insight into how many distinct clients are using your services. It also makes it easier to rate limit specific users, even if they connect from multiple devices or IP addresses.

Authentication is the process of identifying who a user is. From a security standpoint, authenticating clients shrinks the attack surface. If only verified users have access, then there are fewer people who can abuse your services. You can also restrict authenticated users so that they have access to only a limited portion of your APIs, such as to give employees access to administrative functions that normal users shouldn't see.

To authenticate users, sign up for an authentication service provider like Auth0 or Okta or install a service in-house. Authentication services take care of the login process and then give the client an access token, which they can use to gain access to your services. Services like Okta and Auth0 also provide APIs that let you automate signing up new users.

HAProxy sits in front of your APIs and acts as the gatekeeper, making sure that every client presents a valid access token. We described this workflow in the *Authentication* chapter. A client sends their token when making requests to your APIs and HAProxy validates it, checking that it has not expired, that it comes from a

trusted authentication provider, and that it's addressed to the expected recipient.

Because token validation happens at the API gateway, the servers behind the gateway don't need to concern themselves with it. They only need to receive requests and send back responses. The gateway ensures that only users that are allowed to call your APIs are able to do so.

TLS encryption

API messages often carry confidential information that should be protected from eavesdroppers. They also carry a user's secret access token if you've implemented authentication as described in the previous section. The way to protect this confidential information is to enable TLS encryption.

TLS stands for Transport Layer Security and it's the cornerstone of all modern, secure APIs. It enables you to encrypt messages so that they can't be read in-transit, while allowing only the receiver to decrypt them. Enabling it in HAProxy is straightforward. You simply need to upload your TLS certificate and private key to the server and then update your HAProxy configuration to use them to encrypt traffic. We explain the entire process in our blog post HAProxy SSL Termination.

TLS gives you several benefits. First, and perhaps most importantly, it prevents anyone from eavesdropping. Even if an attacker is able to intercept the messages, they won't be able to decrypt them. TLS also maintains the integrity of

the messages, since if an attacker tries to alter random parts of the requests, the client will become aware. It also gives the client a way to validate the server's identity, since only the true server will possess the TLS private key that matches the API's domain name.

By enabling TLS encryption in HAProxy, which acts as a gateway in front of your API servers, you offload that responsibility from your servers. This simplifies your application code and tightens your security by allowing you to store your TLS private key in fewer places.

Rate limiting

Rate limits put a cap on how many API calls a user can make within a period of time. They prevent a single client from overutilizing your application and network resources. After all, even an honest user may write a poorly written client application that makes too many requests. Yet, malicious users intent on flooding your service with requests are all too common. Without rate limits, one misconfigured client or malicious user can quickly overwhelm your services.

In our blog post HAProxy Rate Limiting: Four Examples, we demonstrated how HAProxy supports several types of rate limiting, each with its own set of knobs that you can use to tune the rate limit period, threshold, or fields that are used to identify a user: You can set the period to allow a certain number of API calls per second, minute, hour, or day; You can increase or decrease thresholds dynamically by using

Map files, and you can identify users by a variety of things such as their IP address, URL parameter, or access token.

Rate limits set expectations for your customers about what fair use means. They'll know that they can't hammer your services incessantly, which allows them to plan ahead and weed out any misbehaving client code. Malicious users will be unable to abuse your APIs, for example by calling a function repeatedly to extract your entire catalogue of data.

Anomalous behavior protection

Once you've given a customer access to your APIs, it's important that they use it only in the way that adheres to the terms of your agreement. For example, they should not try to discover or exploit vulnerabilities in your application. They should not try to circumvent your access controls, seek out restricted areas, or try dictionary-style attacks against your API endpoints.

You can detect anomalous behavior by making use of HAProxy's stick tables and ACLs. Stick tables correlate a client's actions across multiple requests, allowing you to verify that they're authenticated, see how many successful vs unsuccessful responses they've received, observe how much data they're uploading or downloading, check how many new connections they're creating, and recognize when they're scanning for vulnerabilities.

ACLs are complementary to stick tables. They're the rules that trigger an action once you detect one of the aforementioned behaviors. For example, you might respond by denying that user's request. Or, you might ban them for a time or simply log the event. We demonstrate many of the options available to you in our blog post Use HAProxy Response Policies to Stop Threats.

HAProxy Enterprise adds more safeguards. With the HAProxy Enterprise version, you can detect bot-like behavior and, through geolocation, check from where in the world the calls are originating. It can also verify that requests are coming from the types of devices you expect through its device detection modules.

Web application firewall

Many of the types of attacks launched against websites—including SQL injection and cross-site scripting—are directed against web APIs too. One way to protect yourself is to introduce a web application firewall (WAF) to inspect all incoming and outgoing traffic and filter out malicious messages.

HAProxy Enterprise ships with its HAProxy Enterprise WAF, which you can run in one of three modes. The simplest of the three modes, SQLi / XSS mode, guards against SQL injection and cross-site scripting attacks alone. This mode is quick to configure and requires very little maintenance, but will give you protection against two of the most common threats against web APIs.

In ModSecurity mode, the WAF blocks a wider range of attacks based on signatures defined in its ruleset files. You get protection against many of the top attack vectors including SQL injection, cross-site scripting, remote file inclusion, and remote code execution. This mode requires more tuning to get right, however it gives you leeway to add rules from various third-party vendors.

When operating in a high-stakes environment consider using the zero trust mode, wherein the WAF blocks all types of requests that have not been explicitly allowed. You might use this to protect APIs that connect critical systems within your organization where you know who the sender and receiver will be and exactly the type of messages they'll exchange.

Protecting your APIs with a WAF is a smart move since it's difficult to guarantee that the applications serving your APIs will always be 100% secure by themselves. A WAF stops attacks at the API gateway before they reach farther into your network.

Bot management

Attackers use bots to launch large, coordinated assaults against APIs. Bots automate the tedium of hacking. They can repeatedly invoke an API function to fetch all of the stored data; They can try endless combinations of values for a function's parameter list, seeking the key that unlocks a vulnerability; They can attempt to inject code into as many endpoints as possible.

A bot does the boring work of hacking so that a human doesn't have to. APIs are easy targets for bots because they are meant to be consumed by software anyway. There's no web UI to get around, no text boxes to find and fill out, no buttons to click. An API is meant to be used by machines.

The challenge with detecting *malicious* bots is that normal consumers of your APIs will also be software programs. Countermeasures like reCAPTCHAs and JavaScript challenges won't be suitable since they would block legitimate software-based clients. However, all clients must follow your usage policy and those that don't deserve to be dealt with in the same way.

HAProxy comes with mechanisms for detecting whether a client is behaving badly. In our blog post Bot Protection with HAProxy, we give examples of bots that make thousands of requests to unique URLs in order to scan your service for vulnerabilities. Other bots will call the same URL many times, such as to try to brute force their way past your authentication mechanisms.

You can discover bad behavior with anomalous behavior detection, as we described earlier. The rate limits we described earlier will also stop clients that make too many requests within a period of time. In the end, when all clients are expected to be machines, you have to fall back to treating them all with the same rules. An API gateway is the best place to enforce these rules.

With HAProxy Enterprise, you can aggregate the stick table data from multiple API gateways. That makes it possible to

identify behaviors even when a bot's requests are routed to multiple nodes in an active/active cluster. This gives you the full picture of the volume and character of the requests.

Conclusion

In this chapter, you learned several ways to protect your APIs using an API gateway. By layering on authentication, TLS encryption, rate limiting, anomalous behavior protection, a WAF, and bot management you can stop attackers before they reach your servers. Having this level of security for your APIs is more important than ever as more types of applications rely on APIs to connect to their data. Having it all in one place means you can provide that security across your entire catalogue of services.

Visit us at https://www.haproxy.com