<https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4>

# Part 1: Rate Limiting: A Useful Tool with Distributed Systems

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----6be2b1a4f5f4--------------------------------)

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----6be2b1a4f5f4--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff94c05083736&operation=register&redirect=https%3A%2F%2Fblog.getambassador.io%2Frate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4&user=Daniel%20Bryant&userId=f94c05083736&source=post_page-f94c05083736----6be2b1a4f5f4---------------------follow_byline-----------)

[Apr 26, 2018](https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4?source=post_page-----6be2b1a4f5f4--------------------------------) · 7 min read

Within the domain of computing, rate limiting is used to control the rate of operations initiated or consumed, or traffic sent or received. If you have been developing software for more than a year or so, you have most likely bumped into this concept. However, as with many architectural challenges, there is usually more tradeoffs to consider than can first appear. This article outlines some of the implementations, benefits and challenges with rate limiting in modern distributed applications.

# Why Rate Limit?

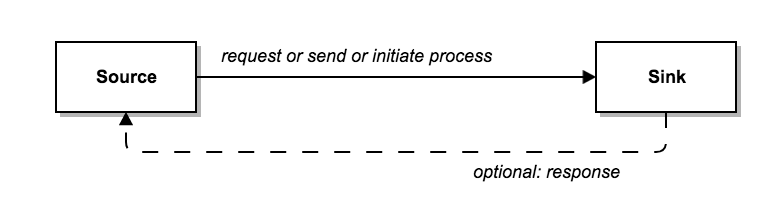
You implement rate limiting primarily for one of three reasons: to prevent a denial of service (intentional or otherwise) through resource exhaustion, to limit the impact (or potential) of cascading failure, or to restrict or meter resource usage.

The denial of service prevention pattern can be seen by organisations like [Twitter](https://developer.twitter.com/en/docs/basics/rate-limiting) or [Ebay](https://go.developer.ebay.com/api-call-limits) placing a rate limiter in front of their SaaS APIs in order to prevent malicious attacks from shutting down the API backends and to provide consistent service for all consumers. Using rate limiting to prevent cascading failure (where some components within your system are partially degraded) can be seen within load shedding policies from payment APIs like [Stripe](https://stripe.com/blog/rate-limiters). The restriction (or metering) usage pattern can be seen when polling an external information source for new data, such as health checking, where we only need to obtain data periodically or we may be charged for each request we initiate.

# What Are Your Options?

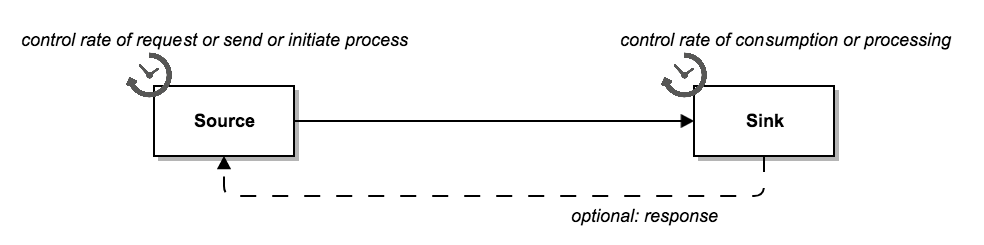
Let’s keep things simple for a moment, and assume that you are dealing with rate limiting within a point-to-point communication model. In this case you can implement rate limiting at either point — on the initiator/sender, the “source”, or on the consumer/receiver, the “sink” — and there are also additional “middleware” options:

https://miro.medium.com/max/30/0*WrOahD1jC0VpJHjT.?q=20



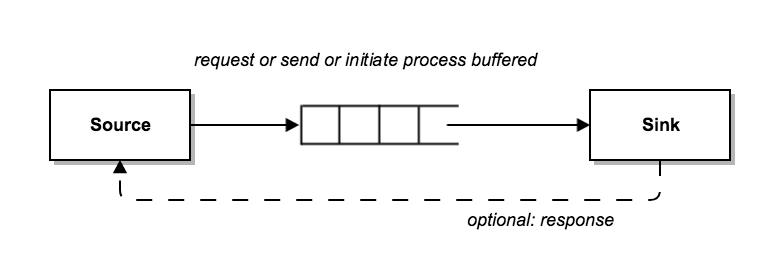
* You can control the rate at which the request is initiated or sent at the source — think a time-throttled loop making a periodic API request
* You can control the rate at which the request is being consumed or received at the sink — think new inbound HTTP connections that are refused until the current task/thread has finished processing

https://miro.medium.com/max/30/1*kxcKGAENnnrgHwRcuF8b7A.png?q=20



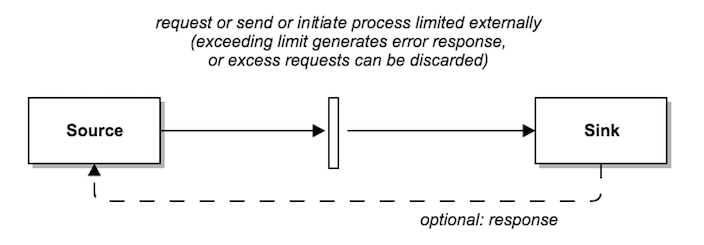
* You can use an intermediary to buffer the initiation or send requests — perhaps placing the request within a queue (priorities can be applied to this queue, providing differing levels of SLA for requests)

https://miro.medium.com/max/30/0*62gSexVvIgDtDBlj.?q=20



* You can use an intermediary to limit the initiation or send requests — perhaps using some form of proxy or gateway that trips a circuit-breaker when the downstream service is not accepting anymore requests

https://miro.medium.com/max/30/0*0nrOQFZO9iPN5S1V.?q=20



# What Are the Tradeoffs?

If you are developing a system and have full control of both points, then all implementation options are available to you, and you simply have to collaborate on the implementations and clearly communicate which points (and corresponding components) have the associated rate limiting responsibilities.

On the flip side, if you only control one of the points — say the sink, or a publicly available API — then your options are somewhat more limited, as you can’t rely on the sources following any guidelines or rules (even if the system contains no bad actors). Even if you do control both points, you may still want to implement a “belt and braces” approach that includes rate limiting at both ends.

Other tradeoffs to consider include:

**The ability of sources and sinks to handle the rate limiting.**

* Sometimes it is not possible to implement effective rate limiting within components, due to programming models or limited resources available etc
* Within a distributed system, rate limiting on an individual component may not provide the required functionality (or at least, not without some level of coordination). For example, if you have rate limited an individual source making calls but you need to horizontally scale to two sources to meet demand, you may now be making twice the allowable calls
* You may also not want backend service engineers writing rate limiting functionality, as this could lead to bespoke implementations or variance between polyglot programming stacks.
* If an application is under heavy or very spiky load, you may want to offload any rate limiting functionality to an external service in order to prevent resources being wasted within the application by performing the rate limiting tasks.
* I’m sure you’ve heard of the single responsibility principle, and at the coarse-grained architecture level, you may require that auxiliary functionality like rate limiting is provided by an external component that has this responsibility

**The failure modes of any rate limiting middleware**

* You will need to determine what happens when a rate limiting service crashes (should the service fail open or closed?), and if the service is buffering requests you may need a restart policy (should requests be buffered to disk?)

**The flexibility of algorithms used by rate limiting middleware**

* The primary advantage of writing rate limiting functionality into a source or sink application that you own is that you have full control over how the rate limiting algorithm is implemented, e.g. [token bucket](https://en.wikipedia.org/wiki/Token_bucket), [fixed window, sliding window](https://blog.figma.com/an-alternative-approach-to-rate-limiting-f8a06cf7c94c) and also the how the request (meta)data is used to make decisions.
* You often have to evaluate which algorithms are available “out-of-the-box” with external rate limiters, and determine whether others (including the associated metadata processing) can be plugged in

# Examples

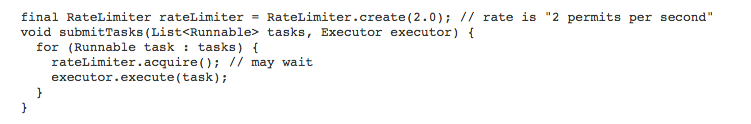
In order to make these ideas a little more concrete, let’s look at a couple of examples.

**Running a task that calls a third-party SDK with call limits or a metered charge per call [you own the source, but not the sink]**

For both the call limits and metered charge sink scenario, I want to implement local (source) rate limiting. I can assume that if I exceed the rate limit, then I may receive an error, or I may get (temporarily) blocked — I would need to confirm the SLA or check the documentation for a production implementation — and whatever happens I don’t want my application simply spinning in a loop constantly attempting calls, as at worse this simply wastes my resources. Without source rate limiting for a metered call, I simply end up paying a lot, and no one like that!

In the Java world, I often use Google’s Guava RateLimiter for this type of problem. An example of the type of code I would write in my (source) application would be:

https://miro.medium.com/max/30/0*ydCsFy-M-qtIWD4M.?q=20

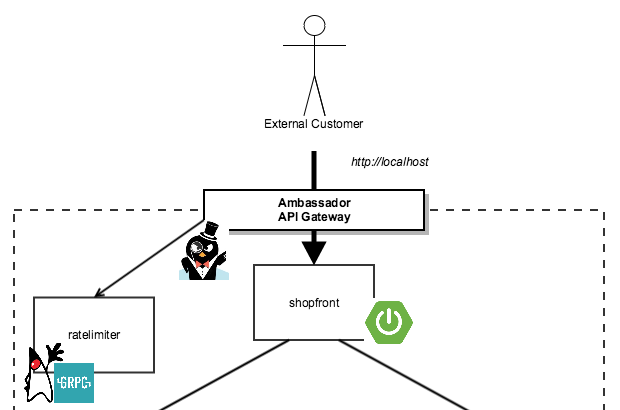


This is a simplified example from the Guava [RateLimiter JavaDoc](https://google.github.io/guava/releases/19.0/api/docs/index.html?com/google/common/util/concurrent/RateLimiter.html), and in reality I would most likely have some Exception handling within the task execution block.

**Offering a public API [you own the sink, but not (all of) the source(s)]**

In this scenario the only way you can guard against the backend of the API being overwhelmed is by rate limiting at the sink, preferably by offloading the limiting responsibility to an external service such as a API gateway.

https://miro.medium.com/max/30/0*QGmzowsbkyFzxGFx.?q=20



# Frequently Asked Questions (2019 Update)

Here are the answers to some frequently asked questions about this article (and rate limiting in general!) since we originally published 18 months ago!

## Why should I rate limit an application or service?

You implement rate limiting primarily for one of three reasons: to prevent a denial of service (intentional or otherwise) through resource exhaustion, to limit the impact (or potential) of cascading failure, or to restrict or meter resource usage.

## Can you explain the fundamental theory and options for rate limiting applications or services?

You can control the rate at which the request is initiated or sent at the source — think a time-throttled loop making a periodic API request.

Or you can control the rate at the sink — think new inbound HTTP connections that are refused until the current task/thread has finished processing.

You can also use an intermediary to buffer the initiation or send requests, perhaps placing the request within a queue. Additionally, you can use an intermediary to limit the initiation or send requests. For example, using some form of proxy or gateway that trips a circuit-breaker when the downstream service is not accepting anymore requests.

## How can I implement rate limiting with microservices or cloud-based applications?

You can implement rate limiting via application code (using an appropriate library or SDK), via a sidecar proxy like Envoy running alongside your service, or if the service is user-facing, via an API gateway like [Ambassador](https://www.getambassador.io/).

# Conclusion

In this introductory article to our three-part Rate Limiting series you have learned about the motivations for rate limiting, and your options and associated tradeoffs. In the next article I’ll dive into more details for implementing rate limiting algorithms for API gateways!

Continue reading the other articles in this four part series:

<https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02>

# Part 2: Rate Limiting for API gateways

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----892310a2da02--------------------------------)

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----892310a2da02--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff94c05083736&operation=register&redirect=https%3A%2F%2Fblog.getambassador.io%2Frate-limiting-for-api-gateways-892310a2da02&user=Daniel%20Bryant&userId=f94c05083736&source=post_page-f94c05083736----892310a2da02---------------------follow_byline-----------)

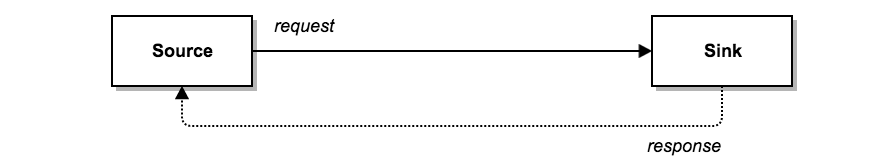
[May 8, 2018](https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02?source=post_page-----892310a2da02--------------------------------) · 7 min read

In the [first article](https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4) of this Rate Limiting series I introduced the motivations for rate limiting, and discussed several implementation options (depending whether you own both sides of the communication, or not) and the associated tradeoffs. This article dives a little deeper into the need for rate limiting with API gateways

# Why Rate Limit with an API Gateway?

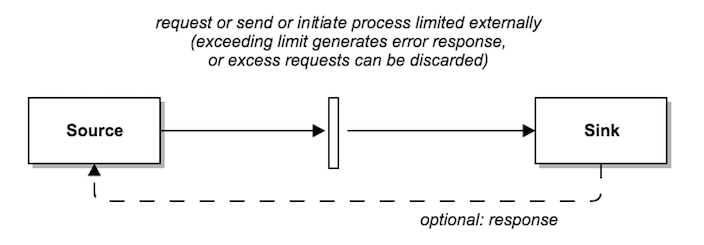
In the first article I discussed options for where to implement rate limiting: the source, the sink, or middleware (literally a service in the middle of the source and sink).

https://miro.medium.com/max/30/0*gPnnpIMRrSVgg9t0.?q=20



When exposing your application via a public API you typically have to implement rate limiting within the sink or middleware that you own. Even if you control the source (client) application you will typically want to guard against bugs that cause excess API request, and also against bad actors who may attempt to subvert your client applications.

https://miro.medium.com/max/30/0*qlW2y6jJxgGH8uwx.?q=20



The Stripe blog has an excellent article on “[Scaling your API with rate limiters](https://stripe.com/blog/rate-limiters)”, which I’ll be referencing throughout this post, and the opening section talks about how rate limiting can help make your API more reliably in the following scenarios:

* One of your users is responsible for a spike in traffic that is overwhelming your application, and you need to stay up for everyone else.
* One of your users has a misbehaving script which is accidentally sending you a lot of requests (trust me, this happens more often than you think — I’ve personally created load test scripts that accidentally triggered a self-inflicted denial of service!). Or, even worse, one of your users is intentionally trying to overwhelm your servers.
* A user is sending you a lot of lower-priority requests, and you want to make sure that it doesn’t affect your high-priority traffic. For example, users sending a high volume of requests for analytics data could affect critical transactions for other users.
* Something in your system has gone wrong internally, and as a result you can’t serve all of your regular traffic and need to drop low-priority requests.

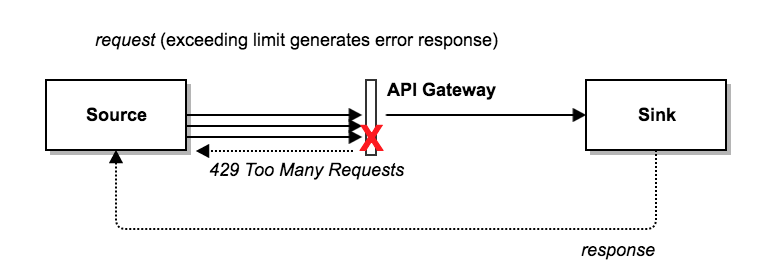
At Datawire we have seen these patterns firsthand, particularly with organisations exposing “freemium” style public API, where it is a clear business requirement to be able to prioritise traffic for paying customer, and also protect against bad actors (intentional or otherwise).

# The Basics of Rate Limiting and Load Shedding

Fundamentally rate limiting is simple. For each request property you want to limit against, you simply keep a count of the number of times each unique instance of the property seen, and reject the associated request if this is over the specified count per time unit. For example, if you wanted to limit the amount of requests each client made you would use the “client identified” property (perhaps set via the request string key “clientID”, or included in the request header), and keep a count for each identifier.

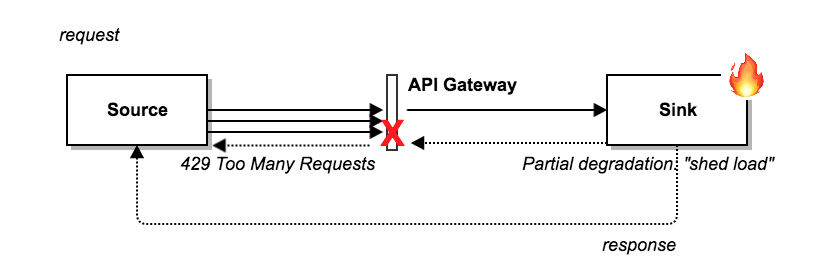
You would also specify a maximum number of requests per time unit, and potentially define an algorithm for how the count is decremented, rather than simply resetting the counter at the start of each unit (more on this later). When a request arrives at the API gateway it will increment the appropriate request count and check to see if this increase would mean that the maximum allowables request per time unit has been exceeded. If so, then this request would be rejected, most commonly returning a [“Too Many Requests” HTTP 429 status code](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429) to the calling client.

https://miro.medium.com/max/30/0*7MkJtV0bgq6dDQtg.?q=20



Closely related to rate limiting is “load shedding”. The primary difference here is that the decision to reject traffic is not based on a property of an individual request (e.g. the clientId), but on the overall state of the application (e.g. database under heavy load). Implementing the ability to shed load at the point of ingress can save a major customer incident if the system is still partially up and running but needs time to recover (or fix).

https://miro.medium.com/max/30/0*YvqaZ6MR77cqYhEH.?q=20



# Challenges with API Gateways

The majority of open source and commercial API gateways offer rate limiting, but one of the challenges with many of these implementations is scalability. Running your API gateway on a single compute instance is relatively simple, and this means you can keep the rate limiting counters in memory. For example, if you were rate limiting on clientId, you would simply check and set (increment) the clientId in an in-memory map with an associated integer counter. However, this approach does not scale past the single instance to a cluster of gateway instances.

I’ve seen some developers attempt to get around this limitation by either using sticky sessions or by dividing the total maximum number of allowable requests by the number of rate limiting instances. However, the problem with this is that neither of these approaches work reliably when deploying and operating applications in a highly dynamic “cloud native” environment, where instances are being destroyed and recreated on-demand, and also scaled dynamically.

The best solution to overcome this limitation is to use some form of high-performance centralised data store to manage the request count. For example, at Lyft, the team use [Redis](https://redis.io/) (presumably run as a highly-available [Redis Sentinel](https://redis.io/topics/sentinel) cluster) to track this [rate limiting data](https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/global_rate_limiting.html) via their Envoy proxy that is deployed as a sidecar to all of their services and datastores. There are some potential issues to be aware of with this approach, particularly around the atomicity of the check-and-set operations with Redis. It is recommended for performance reasons to avoid the use of locking, and both [Stripe](https://gist.github.com/ptarjan/e38f45f2dfe601419ca3af937fff574d) and [Figma](https://blog.figma.com/an-alternative-approach-to-rate-limiting-f8a06cf7c94c) have talked around using the Lua scripting functionality (with guaranteed atomicity) within the Redis engine.

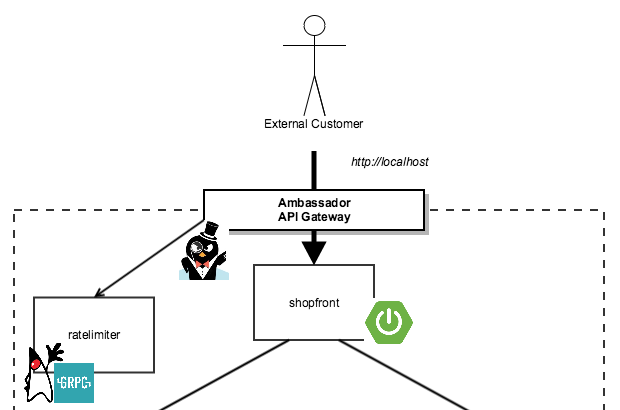
Other challenges often encountered relates to the ability to extract request (meta)data for use in determining the rate limit, and also specifying (or implementing) the associated algorithm used to determine whether a specific request should be rejected. Ideally you would like to be able specify rate limiting in relation to various client properties (e.g. request HTTP method, location, device etc) and the decomposition of your backend (e.g. service endpoint, semantic information such as an user-initiated request vs an app-initiated request, payload expectations).

# Rate Limiting via an External Service

An interesting solution to overcome many of the challenges discussed in the previous section was presented by the [Lyft Engineering team](https://eng.lyft.com/announcing-ratelimit-c2e8f3182555) last year, when they talked about how the Envoy proxy they use as (what we are now calling) a service mesh implements rate limiting by calling out to an external [RateLimit](https://github.com/lyft/ratelimit) service for each request. The Ratelimit service conforms to the Ratelimit protobuf defined [here](https://github.com/lyft/ratelimit/blob/master/proto/ratelimit/ratelimit.proto), which is effectively a rate limit API. The Datawire team has built the open source Ambassador API gateway on top of the Envoy Proxy, and recently [Alex Gervais](https://twitter.com/alex_gervais) has implemented the same [rate limiting support](https://blog.getambassador.io/ambassador-adds-rate-limiting-support-in-0-31-595cc8f91e49) for Ambassador.

As you now have access to a protobuf rate limit service API you can implement a rate limit service in any language you like (or at least any language with protobuf support, which is most modern languages). You also now have complete freedom to implement any rate limiting algorithm you like within the service, and also base the rate limiting decision on any metadata you want to pass to the service. The [examples](https://github.com/lyft/ratelimit#user-content-examples) within the Lyft RateLimit service provide some interesting inspiration! It’s also worth mentioning, that as the Ambassador API gateway runs within Kubernetes, any rate limiting service you create can take advantage of Kubernetes to handle scaling and fault-tolerance.

https://miro.medium.com/max/30/0*C0MXyvXpPDesZ2aI.?q=20



# Wrapping Up with a Look to the Next Article

In this second article of our rate limiting series you have learned about the motivations for rating limiting and load shedding with an API gateway, and you also explored about some of the challenges with doing this. In the final section of the article I presented some ideas around integrating rate limiting witin an API gateway deployed within a modern cloud native platform (like Kubernetes, ECS etc), and discussed how using an external service to do this could allow a lot of flexibility with implementing your exact requirements for a rate limiting algorithm.

Join me for the final part of the series next week, where we take a look at implementing a Java rate limiting service for the Ambassador API gateway (here’s a [sneak peak](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/src/main/java/io/datawire/ambassador/ratelimiter/simpleimpl/RateLimitServer.java) of some of the code!).

In the meantime, please do feel free to email any questions, or jump on the [Ambassador Gitter](https://gitter.im/datawire/ambassador) channel.

Continue reading the other articles in this four part series:

<https://blog.getambassador.io/implementing-a-java-rate-limiting-service-for-the-ambassador-api-gateway-e09d542455da>

# Part 3: Implementing a Java Rate Limiting Service for the Ambassador API Gateway

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----e09d542455da--------------------------------)

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----e09d542455da--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff94c05083736&operation=register&redirect=https%3A%2F%2Fblog.getambassador.io%2Fimplementing-a-java-rate-limiting-service-for-the-ambassador-api-gateway-e09d542455da&user=Daniel%20Bryant&userId=f94c05083736&source=post_page-f94c05083736----e09d542455da---------------------follow_byline-----------)

[May 17, 2018](https://blog.getambassador.io/implementing-a-java-rate-limiting-service-for-the-ambassador-api-gateway-e09d542455da?source=post_page-----e09d542455da--------------------------------) · 11 min read

The rate limiting functionality offered by the Kubernetes-native [Ambassador API gateway](https://www.getambassador.io/) is fully customisable, allowing any service that implements a gRPC endpoint to decide whether a request should be limited or not. In this article, which builds on the previous [part 2](https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02) and [part 1](https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4), you will learn how to build and deploy a simple Java-based rate limiting service for Ambassador.

# Getting Setup: The Docker Java Shop

In my previous tutorial “[Deploying Java Apps with Kubernetes and the Ambassador API Gateway](https://blog.getambassador.io/deploying-java-apps-with-kubernetes-and-the-ambassador-api-gateway-c6e9d9618f1b)” I added the open source Ambassador API gateway to an existing series of Java (Spring Boot and Dropwizard) based services that were deployed into Kubernetes. If you haven’t seen with this, then I would definitely recommend going through this tutorial and the others in the series in order to familiarise yourself with the fundamentals. The rest of this article assumes you’re comfortable building Java-based microservices and deploying them to Kubernetes, and you also have all of the prerequisites installed (I’m using [Docker for Mac Edge](https://docs.docker.com/docker-for-mac/edge-release-notes/), with built-in Kubernetes support, but the principles should be similar if you are using minikube or a remote cluster).

# Prerequisites

You will need to have these installed locally:

* Docker for Desktop — I am using the edge community edition (18.04.0-ce), with in-built support for a local Kubernetes cluster — I have also increased the memory available to Docker to 8Gb, as the Java services can be a little memory-hungry at times :-)
* Editor of choice, Atom or VS code, or IntelliJ for the Java code

You can grab the latest version of the “Docker Java Shop” source code here:

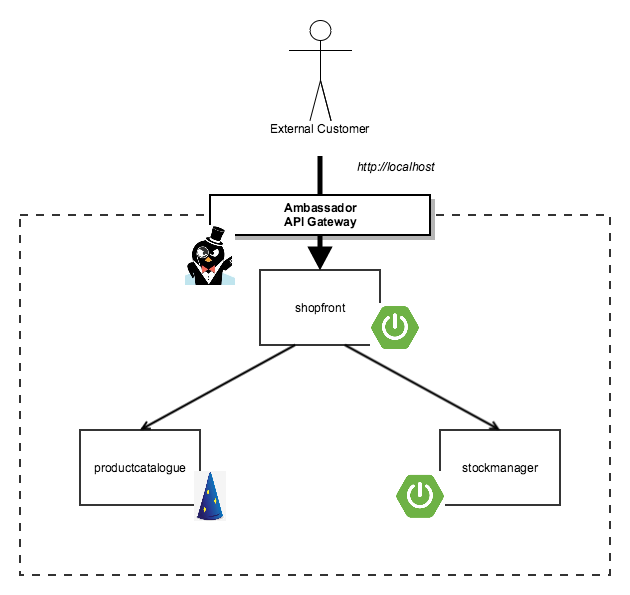
<https://github.com/danielbryantuk/oreilly-docker-java-shopping>

You can clone the repo via SSH like so:

$ git clone [git@github.com](mailto:git@github.com):danielbryantuk/oreilly-docker-java-shopping.git

The initial version of the service architecture and deployment looked as follows:

https://miro.medium.com/max/30/0*SX4TNiV3BK2N3VIw.?q=20



You can see from the diagram that the Docker Java Shopping application consists of primarily three simple services, and in the previous tutorial you added the Ambassador API Gateway as the “front door” of the system. It is worth noting that the Ambassador API Gateway will be running on port 80, the standard unauthenticated web port, and so you will need to make sure there is nothing else locally running on the same port.

# Rate Limiting 101 with the Ambassador API Gateway

I have added a new folder “[kubernetes-ambassador-ratelimit](https://github.com/danielbryantuk/oreilly-docker-java-shopping/tree/master/kubernetes-ambassador-ratelimit)” to the repo that contains the Kubernetes config for this tutorial, and so go ahead and navigate to this directory via the command line. Listing that directory should show the following files:

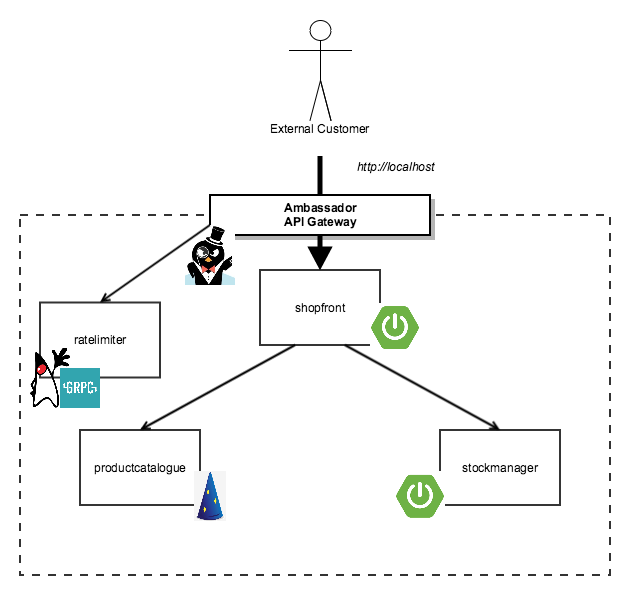
(master \*) oreilly-docker-java-shopping $ cd kubernetes-ambassador-ratelimit/  
(master \*) kubernetes-ambassador-ratelimit $ ll  
total 48  
0 drwxr-xr-x 8 danielbryant staff 256 23 Apr 09:27 .  
0 drwxr-xr-x 19 danielbryant staff 608 23 Apr 09:27 ..  
8 -rw-r — r — 1 danielbryant staff 2033 23 Apr 09:27 ambassador-no-rbac.yaml  
8 -rw-r — r — 1 danielbryant staff 698 23 Apr 10:30 ambassador-rate-limiter.yaml  
8 -rw-r — r — 1 danielbryant staff 476 23 Apr 10:30 ambassador-service.yaml  
8 -rw-r — r — 1 danielbryant staff 711 23 Apr 09:27 productcatalogue-service.yaml  
8 -rw-r — r — 1 danielbryant staff 659 23 Apr 10:02 shopfront-service.yaml  
8 -rw-r — r — 1 danielbryant staff 678 23 Apr 09:27 stockmanager-service.yaml

You can apply these Kubernetes config files with the following command:

$ kubectl apply -f .

Doing so will deploy the following service architecture, with the primary difference from the previous architecture being the addition of the “ratelimiter” service. This service is written in Java, without a web/microservices framework, and it exposes a gRPC endpoint that can be used by Ambassador for rate limiting — this allows for a customisation and flexibility in regards for the rate limiting algorithm you can implement (for more details on the benefits of this, check out my [earlier article](https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02)!).

https://miro.medium.com/max/30/0*AfN0iWZkyp22I6CQ.?q=20



[[https://miro.medium.com/max/30/0*yZRHqy1V0vH6rUZo?q=20](https://www.getambassador.io/products/edge-stack/api-gateway/)](https://www.getambassador.io/products/edge-stack/api-gateway/)

[[](https://www.getambassador.io/products/edge-stack/api-gateway/)](https://www.getambassador.io/products/edge-stack/api-gateway/)

# Exploring the Rate Limiter Kubernetes Service

The ratelimiter service is deployed into Kubernetes just like any other service, and could be horizontally scaled as appropriate. Here is the contents of ambassador-rate-limiter.yaml Kubernetes config file:

---  
apiVersion: v1  
kind: Service  
metadata:  
 name: ratelimiter  
 annotations:  
 getambassador.io/config: |  
 ---  
 apiVersion: ambassador/v0  
 kind: RateLimitService  
 name: ratelimiter\_svc  
 service: "ratelimiter:50051"  
 labels:  
 app: ratelimiter  
spec:  
 type: ClusterIP  
 selector:  
 app: ratelimiter  
 ports:  
 - protocol: TCP  
 port: 50051  
 name: http---  
apiVersion: v1  
kind: ReplicationController  
metadata:  
 name: ratelimiter  
spec:  
 replicas: 1  
 template:  
 metadata:  
 labels:  
 app: ratelimiter  
 spec:  
 containers:  
 - name: ratelimiter  
 image: danielbryantuk/ratelimiter:0.3  
 ports:  
 - containerPort: 50051

You will explore the contents of the underlying “danielbryantuk/ratelimiter:0.3” Docker image later in the article, but for now all you need to know is that this service is running within the cluster, and exposes port 50051.

In the ambassador-service.yaml config file I have also updated the Ambassador Kubernetes annotations config to ensure that requests to the shopfront service are rate limited simply by the inclusion of the “rate\_limits” property. I have also added some additional metadata “- descriptor: Example descriptor”, which I will explain in more detail in the next article. For now, I’ll say that this is a good way to pass additional metadata into the rate limiting service.

---  
apiVersion: v1  
kind: Service  
metadata:  
 labels:  
 service: ambassador  
 name: ambassador  
 annotations:  
 getambassador.io/config: |  
 ---  
 apiVersion: ambassador/v0  
 kind: Mapping  
 name: shopfront\_stable  
 prefix: /shopfront/  
 service: shopfront:8010  
 rate\_limits:  
 - descriptor: Example descriptor

You can check that the deployment has succeeded using kubectl:

(master \*) kubernetes-ambassador-ratelimit $ kubectl get svc  
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE  
ambassador LoadBalancer 10.105.253.3 localhost 80:30051/TCP 1d  
ambassador-admin NodePort 10.107.15.225 <none> 8877:30637/TCP 1d  
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 16d  
productcatalogue ClusterIP 10.109.48.26 <none> 8020/TCP 1d  
ratelimiter ClusterIP 10.97.122.140 <none> 50051/TCP 1d  
shopfront ClusterIP 10.98.207.100 <none> 8010/TCP 1d  
stockmanager ClusterIP 10.107.208.180 <none> 8030/TCP 1d

All six of our services look good to go (plus the Kubernetes service) — that’s three Java services, two Ambassador services, and the ratelimiter service.

You can test the deployment by making a curl to the shopfront endpoint, which (as shown above) should be running on the EXTERNAL-IP of localhost on port 80:

(master \*) kubernetes-ambassador-ratelimit $ curl localhost/shopfront/  
<!DOCTYPE html><html lang="en" xmlns="<http://www.w3.org/1999/xhtml>">  
<head>  
 <meta charset="utf-8" />...</div>  
</div>  
<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->  
<script src="<https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js>"></script>  
<!-- Include all compiled plugins (below), or include individual files as needed -->  
<script src="js/bootstrap.min.js"></script>  
</body>  
</html>(master \*) kubernetes-ambassador-ratelimit $

You will notice that the produces a lot of HTML, which is simply the frontpage of the Docker Java Shop, and can be more easily viewed within a browser pointed at <http://localhost/shopfront/> However, for our rate limiting experiments it will be easier to use curl.

# Testing the Rate Limiting

For this demonstration rate limiting service, I have decided to rate limit simply against the service itself i.e. when the rate limit service calculates whether or not to limit a request, the only metrics I will be considering is the number of requests made against a specific backend service within a time period. The rate limiting algorithm implemented within the code uses the [token-bucket algorithm](https://en.wikipedia.org/wiki/Token_bucket) with a maximum bucket size of 20, and a refill rate of 10 tokens per second. Because the rate limiting is currently associated with any request, this means that you can make 10 requests against the API per second without any issues, and you can also burst above this temporarily because the bucket initially contains 20 tokens. However, as soon as the initial “burst” tokens have been used and you attempt to make more than 10 requests per second, then you will receive an HTTP 429 “Too Many Requests” status code. At this point the Ambassador API gateway is not forwarding the requests to the backend service.

Let’s see if you can simulate this by issuing lots of requests via curl. You’ll want to suppress the HTML payload being displayed ( — output /dev/null) and also the curl request ( — silent), but you still want to see the non-OK HTTP response status codes ( — show-error — fail). You can put all of these curl options together with a simple bash loop and date output (to show what time you are making requests) in order to create a very crude load generator (and get ready to CTRL-C in order to terminate the loop!):

$ while true; do curl --silent --output /dev/null --show-error --fail <http://localhost/shopfront/>; echo -e $(date);done(master \*) kubernetes-ambassador-ratelimit $ while true; do curl --silent --output /dev/null --show-error --fail <http://localhost/shopfront/>; echo -e $(date);done  
Tue 24 Apr 2018 14:16:31 BST  
Tue 24 Apr 2018 14:16:31 BST  
Tue 24 Apr 2018 14:16:31 BST  
Tue 24 Apr 2018 14:16:31 BST...Tue 24 Apr 2018 14:16:35 BST  
curl: (22) The requested URL returned error: 429 Too Many Requests  
Tue 24 Apr 2018 14:16:35 BST  
curl: (22) The requested URL returned error: 429 Too Many Requests  
Tue 24 Apr 2018 14:16:35 BST  
Tue 24 Apr 2018 14:16:35 BST  
curl: (22) The requested URL returned error: 429 Too Many Requests  
Tue 24 Apr 2018 14:16:35 BST  
curl: (22) The requested URL returned error: 429 Too Many Requests  
Tue 24 Apr 2018 14:16:35 BST  
^C

As you can see, the first several requests are served fine, as evident by the date the request was made being displayed alongside no errors, and quickly (at least on my Mac) the loop exceeds 10 requests per second, and I start receiving 429 HTTP response code errors.

As an aside, I would normally use the [Apache Benchmarking “ab”](https://httpd.apache.org/docs/2.4/programs/ab.html) load generating tool for this type of simple experiment, but I believe ab has an issue with calling localhost (or the Docker config was presenting some problems for me).

# Examining the Rate Limiter Service

The code for the Ambassador Java rate limiting service can be found in the repo [ambassador-java-rate-limiter](https://github.com/danielbryantuk/ambassador-java-rate-limiter) on my GitHub account. In this repo you will find not only the code, but also the [Dockerfile](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/Dockerfile) I have used to build the container image that I pushed to DockerHub. Using this Dockerfile as a template, you can make modifications to the code and then build and push your own image to DockerHub. You can then modify the [ambassador-rate-limiter.yaml](https://github.com/danielbryantuk/oreilly-docker-java-shopping/blob/master/kubernetes-ambassador-ratelimit/ambassador-rate-limiter.yaml) file in the main Docker Java Shopping repo to use your service for rate limiting.

# Exploring the Java Code

If you now dive into the actual Java code, the main class of interest is [RateLimiterServer](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/src/main/java/io/datawire/ambassador/ratelimiter/simpleimpl/RateLimitServer.java), which implements the rate limiting gRPC interface defined by the [Envoy proxy](https://www.datawire.io/envoyproxy/) that is used within the Ambassador API — I’ve created a local copy of the [ratelimit.proto](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/src/main/proto/ratelimit.proto) interface that is used by the gRPC Java build tooling defined in the Maven [pom.xml](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/pom.xml). There are three primary points of interest in the code: implementing the gRPC interface, running the gRPC server, and implementing the actual rate limiting code. Let’s now look at these in turn.

# Implementing the Rate Limiting gRPC Interface

If you look into the inner class within RateLimitServer, named “RateLimiterImpl”, which extends RateLimitServiceGrpc.RateLimitServiceImplBase, you can see that I have overridden a method from this abstract class:

public void shouldRateLimit(Ratelimit.RateLimitRequest rateLimitRequest, StreamObserver<Ratelimit.RateLimitResponse> responseStreamObserver)

A lot of the naming conventions used here come from the Java gRPC libraries, and for more information you can consult the [gRPC Java documentation](https://grpc.io/docs/tutorials/basic/java.html). Having said this, you can clearly see the root of a lot of the names if you look into the [ratelimit.proto](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/src/main/proto/ratelimit.proto) file that defines the expected rate limiting interface by the Envoy proxy that is used behind the scenes of Ambassador. For example, you can see that the core service defined in this file is named RateLimitService (line 9), and there is a single RPC method defined within the service “rpc ShouldRateLimit (RateLimitRequest) returns (RateLimitResponse) {}” (line 11) which is implemented in Java through the method signature shown above for “shouldRateLimit”.

If you are interested, a lot of the Java gRPC code generation magic is conducted by the “protobuf-maven-plugin” (line 99 of the [pom.xml](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/pom.xml)).

# Running the gRPC server

Once you have implemented the gRPC interface defined with ratelimit.proto, the next thing to do is to create a gRPC server that can listen and reply to requests made to it. If you look into the content of the [RateLimitServer](https://github.com/danielbryantuk/ambassador-java-rate-limiter/blob/master/src/main/java/io/datawire/ambassador/ratelimiter/simpleimpl/RateLimitServer.java), you can follow the chain of processing from the main method. In a nutshell, the main method creates an instance of the RateLimitServer class, calls the start() method, and then calls the blockUntilShutdown() method. This starts an instance of the class, exposes the gRPC interface on the defined port, and listens for requests.

# Implementing the Java Rate Limiting Code

The actual Java code responsible for the rate limiting process is contained within the shouldRateLimit() (line 75) method of the RateLimiterImpl inner class. Rather than implementing my own rate limiting algorithm, I’m using the popular [bucket4j](https://github.com/vladimir-bukhtoyarov/bucket4j) Java rate limiting library that is based on [token-bucket algorithm](https://en.wikipedia.org/wiki/Token_bucket). As I am limiting the requests made to each service, each bucket will be identified (or keyed) with the service name. Every request to each service will remove a token from the associated bucket. In this example I am not storing the buckets in an external database, and instead have opted to use an in-memory ConcurrentHashMap. If I was implementing this service for a production use case, then I would typically use an external persistence store to allow enable the use of horizontal scalability, probably something like Redis. For now you will have to bear in mind that if you horizontally scale the rate limit service without changing each services bucket limits, then you will be increasing the number of allowable (non-rate limited) requests directly in relation to the increased number of services.

An excerpt of the RateLimiterImpl code that creates the bucket4j bucket can be seen below:

private Bucket createNewBucket() {  
 long overdraft = 20;  
 Refill refill = Refill.smooth(10, Duration.ofSeconds(1));  
 Bandwidth limit = Bandwidth.classic(overdraft, refill);  
 return Bucket4j.builder().addLimit(limit).build();  
}

The shouldRateLimit method code can be seen below, and this simply attempts to tryConsume(1) — try and consume one token from the bucket — before returning an appropriate response code.

[@Override](http://twitter.com/Override)  
public void shouldRateLimit(Ratelimit.RateLimitRequest rateLimitRequest, StreamObserver<Ratelimit.RateLimitResponse> responseStreamObserver) {  
 logDebug(rateLimitRequest);  
 String destServiceName = extractDestServiceNameFrom(rateLimitRequest);  
 Bucket bucket = getServiceBucketFor(destServiceName);Ratelimit.RateLimitResponse.Code code;  
 if (bucket.tryConsume(1)) {  
 code = Ratelimit.RateLimitResponse.Code.OK;  
 } else {  
 code = Ratelimit.RateLimitResponse.Code.OVER\_LIMIT;  
 }Ratelimit.RateLimitResponse rateLimitResponse = generateRateLimitResponse(code);  
 responseStreamObserver.onNext(rateLimitResponse);  
 responseStreamObserver.onCompleted();  
}

The code should be relatively self explanatory, and the primary responsibility of this method is to return either Ratelimit.RateLimitResponse.Code.OK, if no rate limiting is required on the current request; or Ratelimit.RateLimitResponse.Code.OVER\_LIMIT if this request should be denied due to rate limiting. Depending on this response by this gRPC service, the Ambassador API gateway will either pass the request through to the backend service, or short-circuit this trip and simply return a 429 “Too Many Requests” HTTP status code without calling the backend service.

This simple example protects against one service becoming overwhelmed, but hopefully this also demonstrates the core rate limiting concepts, and could be relatively easily adapter to rate limit based on request metadata, such as user ID or something similar.

# Until the Next Time…

This article has demonstrated how you can create a rate limiting service in Java, that can easily be integrated into the Ambassador gateway, and can also be fully customised with any rate limiting logic you require. In the next and final article of the series you will explore the Envoy rate limiting API in more depth, in order to learn more about designing a rate limiting service.

Please feel free to jump on the Ambassador Gitter if you have any questions, or send a tweet over to [@danielbryantuk](https://twitter.com/danielbryantuk/) or [@datawireio](https://twitter.com/datawireio?lang=en)

Continue reading the other articles in this four part series:

<https://blog.getambassador.io/designing-a-rate-limiting-service-for-ambassador-f460e9fabedb>

# Part 4: Designing a Rate Limiting Service for Ambassador

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----f460e9fabedb--------------------------------)

[Daniel Bryant](https://danielbryantuk.medium.com/?source=post_page-----f460e9fabedb--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff94c05083736&operation=register&redirect=https%3A%2F%2Fblog.getambassador.io%2Fdesigning-a-rate-limiting-service-for-ambassador-f460e9fabedb&user=Daniel%20Bryant&userId=f94c05083736&source=post_page-f94c05083736----f460e9fabedb---------------------follow_byline-----------)

[Jun 6, 2018](https://blog.getambassador.io/designing-a-rate-limiting-service-for-ambassador-f460e9fabedb?source=post_page-----f460e9fabedb--------------------------------) · 5 min read

In the [previous part](https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4) of this Rate Limiting article series I showed you how to build and deploy a Java-based rate limiting service that could be integrated with the open source Ambassador API gateway and Kubernetes ([part 2](https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02) and [part 1](https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4) can be found here). Several of you reached out and asked about how best to design a rate limiting service — especially given the unique flexibility of the Envoy proxy rate limiting API that underlies Ambassador — and so this post is will help to address this goal.

# Setting the Scene

If you haven’t looked at part 3 of this rate limiting series, “[Implementing a Java Rate Limiting Service for the Ambassador API Gateway](https://blog.getambassador.io/implementing-a-java-rate-limiting-service-for-the-ambassador-api-gateway-e09d542455da)”, then I would definitely encourage you to do so now (and [part 2](https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02) and [part 1](https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4) will also provide context). The key thing to take away from this is that [Ambassador](https://www.getambassador.io/reference/services/rate-limit-service), much like the [Envoy Proxy](https://www.envoyproxy.io/docs/envoy/latest/api-v1/route_config/rate_limits) that powers it, implements rate limiting by calling out to another service to determine if a request should be rate limited. This is a nice implementation of the separation of concerns pattern (and the single responsibility principle), and as Ambassador is a Kubernetes-native API gateway, then you also get the benefits of deploying the rate limiter as a standard Kubernetes service that is managed by the platform in regards to fault tolerance, and can easily be scaled.

The rest of this post assumes that you have successfully deployed Ambassador to your Kubernetes cluster, and that you have also deployed a rate limiting service as I demonstrated in my previous Medium post. This is what the Kubernetes config for the Java-based rate limiting service looks like:

---  
apiVersion: v1  
kind: Service  
metadata:  
 name: ratelimiter  
 annotations:  
 getambassador.io/config: |  
 ---  
 apiVersion: ambassador/v0  
 kind: RateLimitService  
 name: ratelimiter\_svc  
 service: "ratelimiter:50051"  
 labels:  
 app: ratelimiter  
spec:  
 type: ClusterIP  
 selector:  
 app: ratelimiter  
 ports:  
 - protocol: TCP  
 port: 50051  
 name: http---  
apiVersion: apps/v1beta2  
kind: Deployment  
metadata:  
 name: ratelimiter  
 labels:  
 app: ratelimiter  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: ratelimiter  
 template:  
 metadata:  
 labels:  
 app: ratelimiter  
 spec:  
 containers:  
 - name: ratelimiter  
 image: danielbryantuk/ratelimiter:0.3  
 ports:  
 - containerPort: 50051

# Descriptors

The rate limiting flexibility within Ambassador comes from the ability to specify descriptors and headers on the Kubernetes config, which will be passed through to the rate limiting service. I find it easier to talk about these concepts with an example at hand. Let’s look at a sample Ambassador config for my shopfront app that you have explored in the previous Medium posts:

---  
apiVersion: v1  
kind: Service  
metadata:  
 labels:  
 service: ambassador  
 name: ambassador  
 annotations:  
 getambassador.io/config: |  
 ---  
 apiVersion: ambassador/v0  
 kind: Mapping  
 name: shopfront\_stable  
 prefix: /shopfront/  
 service: shopfront:8010  
 rate\_limits:  
 - descriptor: Example descriptor  
 headers:  
 - "X-MyHeader"  
 - descriptor: Y header descriptor  
 headers:  
 - "Y-MyHeader"

You can see from the rate\_limits config that we have two elements in our YAML list, each with different descriptor values and header lists. As mentioned in the [Ambassador Rate Limiting](https://www.getambassador.io/user-guide/rate-limiting-tutorial#2-configure-ambassador-mappings) docs, if headers are defined then they must be part of the request in order to be rate limited. So, with this example:

* A request made to the shopfront service with no headers will not be eligible for rate limiting (i.e. a call against the rate limiting service that is defined elsewhere in the Ambassador config will not be made)
* A request made to the shopfront service with the header “X-MyHeader:123” will be eligible for rate limiting. The rate limiting service will receive the descriptor information (as a “[generic\_key](https://www.envoyproxy.io/docs/envoy/latest/api-v1/route_config/rate_limits#generic-key)”) associated with the rate\_limits element that matches the “X-MyHeader” header — in this case “Example descriptor” — and also the header key and value e.g. the rate limiting service will receive this request metadata: [{“generic\_key”,“Example descriptor”},{“X-MyHeader”,”123”}]
* A request made to the shopfront service with the header “Y-MyHeader:ABC” will be eligible for rate limiting. The rate limiting service will receive the descriptor information (as a “[generic\_key](https://www.envoyproxy.io/docs/envoy/latest/api-v1/route_config/rate_limits#generic-key)”) associated with the rate\_limits element that matches the “Y-MyHeader” header — in this case “Y header descriptor” — and also the header key and value e.g. the rate limiting service will receive this request metadata: [{“generic\_key”,”Y header descriptor”},{“Y-MyHeader”,”ABC”}]

The decision to rate limit a request, or not, is made within your rate limiting service, and you simply return an appropriate value as specified in the Envoy [ratelimit.proto](https://github.com/envoyproxy/envoy/blob/master/source/common/ratelimit/ratelimit.proto) gRPC rate limit service interface: OK, OVER\_LIMIT or UNKNOWN. Using the descriptor and header combination as described above means that you have two places where you can add request metadata that can be used within the rate limiting service. The descriptor can be added to the Ambassador Kubernetes config at deploy time, and the headers can be added at runtime.

# Working with Example Rate Limiting Metadata

Let’s now look at an example. Say your organisation has created a mobile app that talks to a backend service that is fronted by the Ambassador API gateway, and you want to rate limit requests differently for regular and beta users, and you also want to rate limit unauthenticated users completely different. You have access to UserID and UserType data that could be added to the header of any request:

---  
apiVersion: v1  
kind: Service  
metadata:  
 labels:  
 service: BackendService  
 name: BackendService  
 annotations:  
 getambassador.io/config: |  
 ---  
 apiVersion: ambassador/v0  
 kind: Mapping  
 name: backend\_app  
 prefix: /app/  
 service: backend\_app:8010  
 rate\_limits:  
 - descriptor: Mobile app ingress - authenticated  
 headers:  
 -"UserID"  
 -”UserType”  
 - descriptor: Mobile app ingress - unauthenticated

Any request made with headers “UserID” and “UserType” present will be forwarded to the rate limit service, which will contain the header keys and values alongside the (generic\_key) descriptor value “Mobile app ingress — authenticated”. Requests without these headers will be caught by your second descriptor, and these will be forwarded to the rate limit service will only the (generic\_key) descriptor value “Mobile app ingress — unauthenticated”. Your rate limiting service can then implement an algorithm (in any language you require) to deliver the appropriate behaviour.

# Conclusion

If you are looking for [inspiration](https://eng.lyft.com/announcing-ratelimit-c2e8f3182555), or a [ready-made Ambassador rate limiting service](https://github.com/lyft/ratelimit), then be sure to check out the Envoy documentation and Lyft GitHub repository. In particular, the Lyft [ratelimit](https://github.com/lyft/ratelimit) reference implementation of an Envoy rate limiting service is very useful, both as a drop-in solution, and also as a guide on how any custom rate limiting solutions can load configuration and runtime.

You can find the tutorial on installing the Ambassador API Gateway within Kubernetes and configuring rate limiting within the previous post “[Implementing a Java Rate Limiting Service for the Ambassador API Gateway](https://blog.getambassador.io/implementing-a-java-rate-limiting-service-for-the-ambassador-api-gateway-e09d542455da)”. As usual, you are welcome to post any question via the [Ambassador Gitter](https://gitter.im/datawire/ambassador) channel.