fastly

6 ways to enhance your microservices at the edge

The advantages of adopting microservices over a monolithic architecture are well documented, and their popularity is on the rise. As a form of services-oriented architecture, microservices are not necessarily appropriate for every company. However, breaking application logic into smaller functional services can provide significant benefits, such as greater scalability, modularization, and deployment speed.

Less is known about the nascent but rapidly growing practice of utilizing logic at the edge to strengthen and support microservices architectures. The ability to obtain granular and immediate control by supporting custom logic at the edge of your network is particularly useful in routing traffic to the microservices that make up your product. As an edge cloud platform running on a globally distributed Varnish cluster, Fastly gives you this power. Using Varnish Configuration Language (VCL) we allow you to implement rules that can manipulate requests and responses for improved performance and origin offload.

In this brief, we will discuss six best practices for advancing your microservices by moving decision making to the edge and we will also explain how they work using Fastly's edge cloud platform. Even if you are not a Fastly customer but use Varnish internally, the following approaches may still be useful: presenting a unified domain, defining origin services, routing requests, A/B testing, authenticating, and renaming URLs.

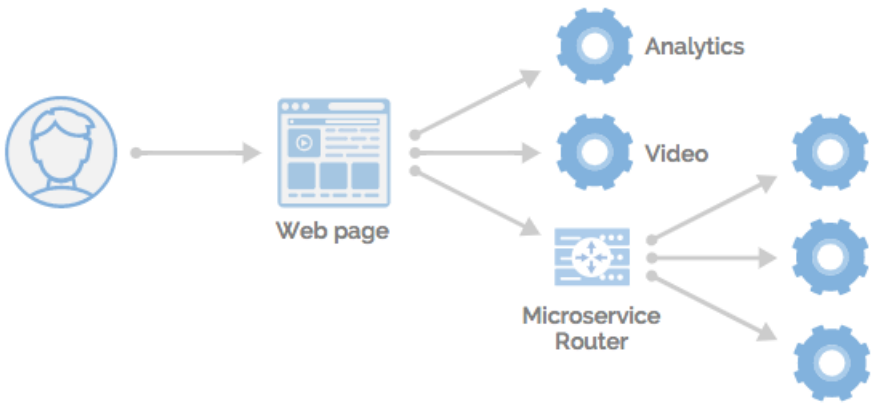
Edge decision making

Fastly's edge cloud platform gives you the power to write and instantly deploy custom VCL, which functions more like a programming language than a configuration file. This allows us to move logic and functionality closer to your users at the network edge. Our programmability, coupled with the highly transparent and real-time nature of our platform, allow us to serve as an extension of your infrastructure. Adding logic at the edge can be highly beneficial to your microservices architecture.

1. Moving towards a single domain

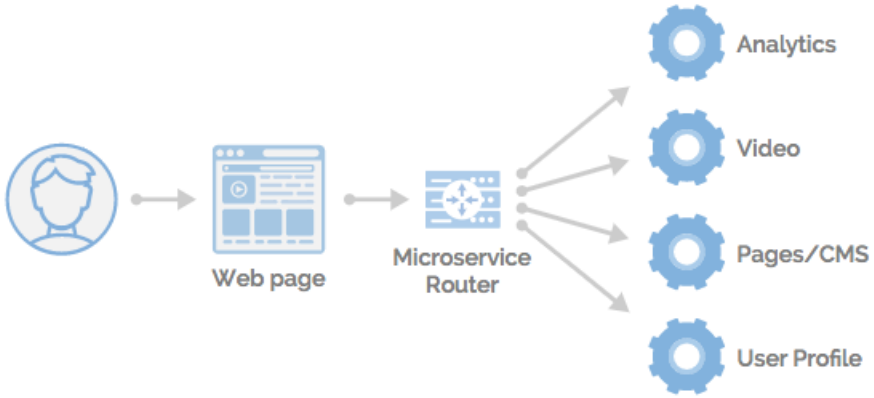
In a typical microservice architecture, multiple similarly-structured services exist behind a routing layer that combines all the routes handled by all the individual services under a single public domain. However, the sheer range of different types of origin services that can be put together in this way is often not fully recognized, leaving some services to remain separate from those hidden behind the router (see figure 1).

Figure 1



Nowadays it makes sense to take an aggressive approach to reducing the number of domains that are involved in loading each web page. Even with HTTP/2 and the rise of TLS encryption, opening a connection to different domains remains relatively expensive. This is the case even if each microservice and/or different services are on a different domain. At the same time individual requests over an existing connection are now largely non-blocking, asynchronous, extremely cheap and allow better optimization of TCP's congestion control mechanism. Additionally, placing your router close to the end user (i.e., on an edge cloud platform) enables you to negotiate the initial cold-start connection very quickly and take advantage of pre-warmed links from the router to each of the origin services. Consider not only application services you've built in-house, but also static storage systems and even third-party services — these too could potentially be placed behind your router. You may even be able to serve the entire page from behind a single domain (see figure 2).

Figure 2



Fastly makes it possible to present a unified API domain to the client by routing microservices from the edge for every flavor of application backend (Heroku and Amazon EC2 are popular choices), as well as static storage services like Google Cloud Storage and third-party tools like Sentry, Google Analytics, and Brightcove.

2. Defining origin services

With a large number of varied backend services, you may also have quite different designs, necessitating a flexible approach to defining how they are routed in the CDN layer. Some backends may handle their own load balancing and expose only one hostname, while others may want traffic balanced across multiple hostnames. Still others may need URL elements to be rewritten slightly to make sense in your overall URL scheme.

One approach used by Fastly customers is to create or generate a service registry at the edge: an always up-to-date data structure that contains a list of all backend services and how they differ. A service registry might look like this:

```
[
  {
    "name": "pages_service",
    "paths": [
    "^/(section|article)/.*",
    "^/api/content/.*"
   ],
   "hosts": [ "pages01.example.com", "pages02.example.com" ],
   "ssl": true,
   "healthcheck": "/healthcheck"
},
{
   "name": "assets_store",
   "paths": [
   "^/resources/.*"
   ],
   "hosts": [ "assets-example-com.eu-west-1.s3.aws.com" ],
   "ssl": true,
   "healthcheck": "/healthcheck"
},
{
   "name": "video_external",
   "paths": [
   {"src": "^/videos/stream/([a-z0-9]+)/?", "dst": "/$1"}
  ],
   "hosts": [ "brightcove01.brightcove.com" ],
   "ssl": true,
   "healthcheck": "/healthcheck"
  },
  …
]
```

The first example above might be an in-house application service in NodeJS or Ruby, running on AWS EC2. It does not do load balancing itself so we need to balance traffic between pages01.example.com and pages02.example.com. The second example could be the static assets, images, styles, fonts etc., hosted on Amazon S3. And finally we can define a backend that belongs to a third party, specifying some URL rewriting so that we can fit the supplier's service design into our own URL scheme.

Of course it's also possible to set up all this routing manually by writing verbose VCL, but Fastly customers will often take a data structure like this and use it to build routing logic in VCL, using an automation process in a CI tool such as CircleCI or Jenkins.

Using Fastly, you can also make use of dynamic server pools to create groups of servers that can be load balanced, dividing the routing process into two logical steps. First, based on the request URL, determine from the edge which service should handle the request (allocate the request to a pooled backend). Second, choose a specific backend host from the allocated server pool. Normally a random process with a uniform distribution is used for that, although a variety of strategies can be implemented, including gradually ramping traffic up and down and A/B testing (see point 4 on page 5).

3. Request routing

Mapping an inbound request to an origin backend is at the heart of a microservice architecture, and the more flexible you can be with this, the more services you can likely absorb into one single external domain. This is best accomplished at the edge, where you can quickly and efficiently direct requests to the appropriate backend.

A common way to define these request-to-origin relationships is via regular expression replacement. Fastly customers can do this at the edge using VCL, employing a workflow such as:

```
if (req.url ~ "^/videos/stream/([a-z0-9]+)/?") {
  set req.backend = pages_service;
  set req.url = regsub(req.url, "^/videos/stream/([a-z0-9]+)/?", "/$1");
  set req.http.Host = "brightcove01.brightcove.com";
} else if (...) {
```

This demonstrates a very specific routing rule, where it will only go to the backend if the video ID part of the URL is an alphanumeric string. If it includes an underscore or other punctuation characters, it won't match this route.

This is good in a way because the microservice receives fewer garbage requests, and it reduces the attack surface area in case your microservice has a vulnerability that can be exploited by crafting a malicious URL. However, it's also common to want the flexibility of just carving out a branch of URL space below a prefix and allocating that to a microservice.

Imagine that you want everything under **/myAccount** to go to the user_profile service. In this kind of situation a corner case crops up when the user requests exactly **/myAccount** without a trailing slash, which would map to an empty path on the backend microservice. Here you can choose whether the router fills in a/ transparently and the user gets back a page as if they had added the / themselves, or alternatively the router could issue a redirect to prod the user onto the right URL. Here's how a Fastly customer might do that in VCL:

```
if (req.url ~ "^/myAccount([^?].*)?") {
  set req.backend = user_profile;
  set var.tmpUrl = regsub(req.url, "^/myAccount([^?].*)?", "$1");
  if (var.tmpUrl == "" && req.url ~ "[^/]$") {
    set var.tmpUrl = req.url "/";
    error 301 var.tmpUrl;
  }
  set req.url = var.tmpUrl;
  set req.http.Host = "user_profile.example.com";
}
```

In VCL, errors terminate requests early so they are a good way to perform redirects from the edge.

The routing layer is also a good place to do request normalization, to reduce the granularity and noise of requests hitting backends. For example, some services may accept query strings, while others don't, so where they don't, consider stripping any query params at the edge. Sorting query params alphabetically can also improve caching performance. Fastly offers querystring related VCL functions that make it easier to manipulate query data.

We can also normalize for variation data. Headers like Accept, Accept-Encoding and Accept-Language (and in the future, Client-Hints and others) are commonly used to send responses that are more tailored to specific client needs, but the requests can be very noisy. Fastly automatically normalizes Accept-Encoding and provides functions to help normalize Accept-Language if you want to do that. For example, the Vary header is one of the most powerful HTTP response headers.

Some backends might be interested in making decisions based on the geolocation of the user, or you might allow this to be part of your routing decisions. If you can discover the physical location of the user using tools at the edge, you can then either directly make use of that data in routing, or add it as additional HTTP request headers so that backend services can use the data without having to determine it for themselves. Fastly provides many related VCL features, such as geolocation and device detection, so that our customers can use geo in a way that suits their use case.

4. A/B testing and ramping

There are cases when you may want to bring in a microservice to gradually replace another or use one service with one group of users and a different service with another group. This is easily achieved without interrupting service to users by using Fastly to add a bit of logic at the edge. Fastly customer Reddit has used such a technique to migrate to a new profile page design. While Reddit describes a way to do this manually, we can also apply this principle generically if we have many microservices and regular requirements for scaling them. The concept of A/B testing builds on the routing logic detailed in the above section. Only a couple of additional requirements are necessary:

- Allow multiple services to claim the same URL space
- Allow routes to be conditional on something other than the URL, such as a dice roll

There are a number of ways of modeling these relationships at the edge. A simple example would be to separate the list of services from the set of global routes, and have a table to map the routes to the service definitions. Below, we are directing all traffic that requests /myAccount to the user_profile service, but dividing the /pages/articles requests between two services in a 90/10 split.

```
{
  "^/myAccount([/\?].*)?": "user_profile",
  "^/pages/articles([/\?].*)?": [
    {"service": "articles_app", "traffic_pc": 90},
    {"service": "articles_app_new", "traffic_pc": 10}
  ]
}
```

Fastly customers could translate this JSON document to VCL incorporating our randomness related VCL functions. Various other techniques utilizing Fastly can be found in our blog post on A/B testing at the edge.

5. Authentication

It's common that multiple backend services might need to hook into the authentication state of the user. With Fastly you could consider reading the authentication cookies at the edge, validating them and translating the data into additional trusted HTTP headers, such as:

MyApp-Auth-State: authenticated
MyApp-Auth-User-Role: subscriber
MyApp-Auth-User-ID: 12345

This avoids the need to perform authentication in every service. At the core of this solution is the ability to compute cryptographic hashes in your edge logic. Fastly has a suite of crypto and hashing related functions that enable these patterns for our customers. Nikkei is one such Fastly customer that uses this technique.

You must decide whether to just do authentication at the edge and pass all requests to backends regardless of authentication state, or whether to also do authorization at the edge (i.e., actually reject requests if the user is not allowed to access the requested resource, without even passing the request to a backend service). Performing **authorization at the edge** will shield your origin from unauthorized client requests, reducing traffic and the corresponding infrastructure cost.

However when building your edge configuration in such a way, you need to know whether a given service has authorization constraints (e.g., does it accept requests from authenticated users or only from paying subscribers?). It may be good enough to apply those constraints to a whole service, or you may choose to allow distinct authorization constraints at the path level.

6. Vanity URLs and URL renaming

While designing your overall URL structure, it makes sense to keep each microservice relatively contained to a coherent branch of the path tree. However, there are some occasions where you really need a top-level, easy-to-remember URL for something. News organizations often make use of these so-called 'vanity URLs' to create paths such as **/world**, **/tech**, and **/politics**. To avoid having huge lists of paths in a routing table, it can be clearer to have a simple key-value map that translates these vanities into 'real' URLs to which you can then apply routing.

Fastly supports this kind of solution via static VCL tables and dynamic Edge Dictionaries, which give you the ability to create dictionaries (key/value pairs that your VCL can reference) inside your Fastly services. VCL tables can be pushed with your versioned configuration, whereas Edge Dictionaries can be instantly updated via API without versioning your configuration.

The edge is just the beginning... for microservices

Using edge logic to strengthen and support your microservices is helpful when it comes to the resiliency and usability of your product. The ability to define, route, A/B test, authenticate, and rename URLs are examples of what you can do at the edge to add functionality and increases performance. Leverage the power of Fastly's edge cloud platform to implement logic that makes the most sense for your architecture.

You can also take advantage of our delivery and security offerings that are built on the core Fastly platform and part of our single high-performance network. Fronting certain microservices (such as those related to authorization and accounts) with TLS delivery, DDoS protection, and WAF will help keep them protected and performant. We also stream 100% of your logs to the edge in real time as part of our standard service, giving you full visibility into the performance and usage of your microservices. With Fastly, the edge is just the beginning for what you can accomplish with microservices.

Go to www.fastly.com/signup to register for a free account and try Fastly for yourself.