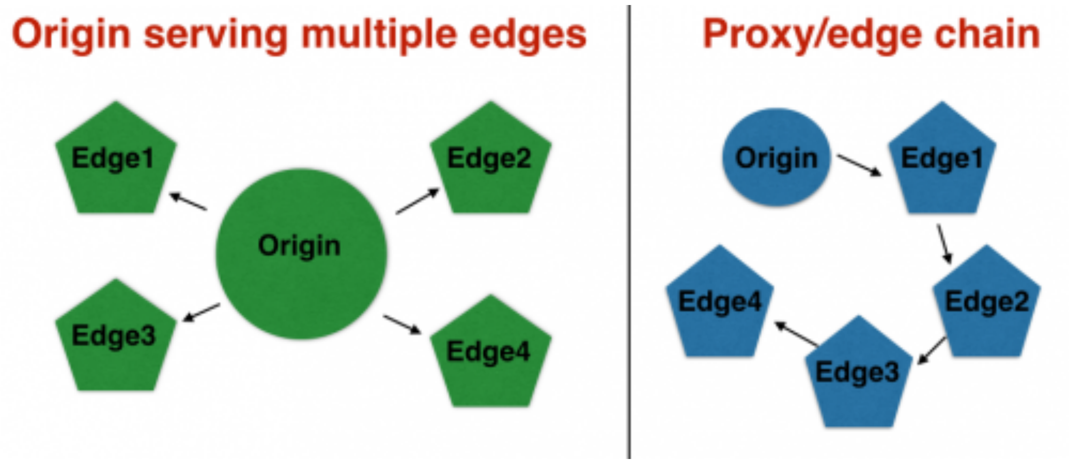
Making your own CDN/Edge Cache on the cheap!

January 28, 2014

I was originally going to split this into a 3-part series, but decided that maybe people would prefer to have everything in one place. So it's all on one page, and you won't be subjected to a pile of ads until the end. Here goes...

Part 1 – Overview

There are numerous ways in which you could roll your own CDN – I'm going to take you through one, with a couple variances in implementation. To keep you from reading 10 paragraphs only to find out "this isn't what I had in mind", here's an image to show what we're pushing towards:



Either of these will work (switching between them is easy), but we'll get into that later.

Costs

Getting a rough idea of your costs (in addition to the domain itself and hosting on the origin server), there are 2 primary costs:

Cost #1 – DNS: For this to work, you're going to need DNS that supports geotargetting your IPs so that visitors are directed to your edge server closest to them. In an ideal world you'd just be able to anycast your IPs, but that ability is out of reach for most people.

There are a couple cost effective options I'll go more thoroughly later: Amazon's Route53, and Rage4. There are likely others, but these were the 2 I found that seemed to be the cheapest. I'll give you a rough idea as to the pricing first though:

Amazon will charge your $0.50/domain/month plus $0.75 per million queries. So you're looking at about $6/year for each domain plus the queries which are pro-rated based on what you've actually used. For a fairly low-traffic site, you could be looking at under $1/month. The DNS servers themselves aren't available over IPv6, though there is support for IP6 (AAAA) records. You're also limited to targeting the "regions" that Amazon uses, which means you'll probably want your edge servers near those geographical regions, but I'll go into that later.

Rage4 will host your domains for free. And the first 250,000 queries per month are free. And the DNS resolvers are dual-stack (IP4 and IP6). And you can target a lot more locations that Amazon. For a low-traffic site, your cost for all this could be $0. If you end up above that 250,000 queries per month, you are looking at 2 EUR (about $2.75 USD currently) up to the first 1 million and then 1 EUR per million after that.

Cost #2 – Hosting for each edge server: While my research has shown it should be *technically possible* to use a few shared hosts as edges (making use of mod_proxy and mod_cache if enabled and possibly mod_ssl for SNI if they're using at least CentOS 6 and WHM 11.38 or so…), you'll probably have a hard time finding a shared host with a suitable configuration. Chances are that you'll want to use some cheap "low end" VPS's for these, located at various locations with nginx thrown on, or perhaps Varnish if you don't need the SSL. LowEndBox is one of the better places to look for those, though you could sift through the offers on WebHostingTalk as well. Even a 128MB VPS will work fine, and you should be able to find a number of them in the US for $10-20/year (after various coupon codes) if you're savvy. Technically you can often find some for around $5/year (and nginx will do fine on 64mb of RAM), but hosts who offer those prices tend to run into challenges. In any case, rounding up, you're realistically looking at $1-2/month each.

Going for numerous low-end boxes does leave an interesting question… how do you automatically route around them if one should go down? That's where Route53 and Rage4 come into play, and they each deal with this differently. Amazon's Route 53 allows you to set up "Health Checks" at $0.75 per server monitored per month, and it will check your server from 3 locations every 30 seconds, and "drop" one from the pool if it goes down, until it's back up. Rage4 on the other hand relies upon Uptime Robot, which checks your servers every 5 minutes (but is free), and will fail-over to another IP address when one isn't responsive. More about both of those later.

So if we ballpark $24/server per year (which is honestly higher than you should be paying) and going the Amazon Route, you're looking at a monthly cost of ($2 x servers) + ($0.50 x domains) + ($0.75 * total DNS queries in millions). For 3 servers (say, US East, US West, and Europe) hosting 1 domain with a million queries per month would be $6 + $0.50 + $0.75 = $7.25/month. If you use health checks, it adds another $2.25 for the 3 servers bringing it to $9.50 per month. Each domain you add to that only adds $0.50 plus additional DNS queries. Not bad.

Going the Rage4 route instead with 3 servers, you're looking at ($2 x servers) = $6/month for low traffic (under 250,000 DNS queries). Assume ($2 x servers) + $2.75/month DNS = $8.25/month for medium traffic (up to 1million DNS queries). Each domain only adds to the cost of traffic.

Cost Summary: You can see that adding extra edge servers has the highest effect on cost, and that adding additional websites (domains) has very little additional cost in comparison. If you had 20 websites in the above cases (instead of just 1), you would be ballparking under $1/month (plus traffic) per website using Amazon, or under $0.50/month (plus traffic for any sites over 250,000 queries) per website using Rage4. It starts to get very cost effective when you look at it that way, especially compared to the $20+ you often have to spend *per site* with other providers.

Amazon Cloudfront vs Cloudflare vs Incapsula vs others vs YOU

If the DNS stuff has you wondering whether going further will be worth it or not, here are a few things you can get by home-rolling that you can't get with the others, or that you can do more cheaply than the others:

SSL support – Adding SSL to your home-rolled CDN is virtually free. StartSSL will give a DV cert for free (or you can spend under $10/year from other providers) and you can plop that onto all the edge servers you run – it only costs you in time. Amazon has their own wildcard SSL built into the *.cloudfront.net addresses (free) but if you want anything at .yoursite.com it's $600/month. Cloudflare/Incapsula don't offer any free SSL options, but since they go in front of your domain, they're able to provide free DV SSL on your domain in their paid packages. For Cloudflare that means $20/month (+$5 for additional domains), and for Incapsula $19/month. To step up to the "Amazon" level where you use your own certificate, you're looking at the $200/month plan from CF or $299/month from Incapsula.

IPv6 – Of the 3 listed above, CloudFlare is the only one that supports it (and truthfully their implementation is more flexible than what you'll likely implement). But most others don't serve over IPv6. On the other hand, IP6 is fairly easy to implement when rolling your own.

Long time in the edge cache for static resources – Here's the biggy. Even if you've got your mycat.jpg expiry set to 50 years, a number of edge caches / CDN's (including the above) will kick it from an edge cache within 24-48 hours if it's not frequently accessed, at which point the next person who visits your site from that location gets a nice long wait while it's fetched from your origin. The conditions as to whether it's kicked or not aren't generally advertised, but it's safe to say that if a number of your older pages only get a few visits per day from various locations around the world, chances are they're all waiting *longer* than they would if you didn't use a CDN, because each edge ends up asking the origin for a fresh file resulting in extra overhead. CDN's are great for high-traffic sites, but low-medium traffic sites that are otherwise well-tuned will often see an overall *increase* in page load time for their users. Fortunately, in your home-rolled CDN you can keep items in the cache for as long as you want.

So by home-rolling, you can have anything on your site hosted retained on all your "edge caches" for as long as you want. Years, even (not that you want that, but maybe for your favicon….). That's not possible with Amazon, Cloudfront, Incapsula, and most others – at best your frequently accessed stuff might stay in their caches for long periods – but infrequently accessed stuff gets shafted.

Part 2 – Getting started…

Decide whether you're going to just cache images/css/js/etc (common "static" stuff), or your entire site. If the former, note that you'll be using a subdomain for it, and you'll need to keep your resources on that subdomain. If the latter, you're just using your regular domain which is quite a bit easier.

First, you're going to need the origin host. It could feasibly be a regular old shared host, although you're best to set all your desired caching behaviour (read: expires times) from the origin, so make sure it's suitable – if you're using WordPress on Apache, a lot of caching plugins will set up expires headers for you, and caching plugins should exist for most other CMS's too. Otherwise you can set up the behaviour manually via .htaccess or an nginx configuration file.

Use Chrome/Firefox/etc to check the headers for all your pages before you start setting up the edge servers, to make sure cache/expiry times look correct. I prefer using "Cache Control" headers with max-age and public, though you could use "Expires" headers instead.

> The downside to Expires is that objects are given a finite date/time at which point they should be retrieved again, which means if you've set an object to cache for 24 hours and you're into the 23.5 hour mark, the edge cache is serving these to clients with only 0.5 hours remaining, which the client will respect. So suddenly when that final moment hits, all your visitors will be hitting the edge cache for the new version. Now that *could* be what you want. But if it isn't, you'll want to check out cache control.
>
> Cache Control essentially states "cache for x seconds starting the moment you get this", so it's a little more lenient. Even if your edge cache is a few seconds away from needing to grab the fresh page, a visitor who hits the edge before the deadline will cache for the full allotted time (24 hours in this example). The downside here is that in a long chain of proxies, someone could end up with very, very old content. Imagine the case where proxy1 has had the content for 23 out of 24 hours, and proxy2 grabs from it – proxy2 now holds that page for 24 hours. If 23 hours later proxy3 grabs from proxy2, proxy3 will now hold it for 24 hours. So a visitor who hits proxy 3 before *that* 24 hours is up now has content that is almost 70 hours old! That might be fine for images, but if you're caching your dynamic content too, that might be a problem.

Determine your desired behaviour and go from there. Assuming your headers look good, it's time to set up your first "edge cache", or "reverse caching proxy" server. I'll assume you've muddled through installing NGINX on a VPS for this, and just need something to plug into your config. Here are a few chunks you can plunk in where applicable, though **I suggest trying to work out a config yourself and just come back to use these as references if you get stuck**:

```
#
# If you are using your edge servers for multiple domains and
# also to serve other sites (locally hosted on this edge),
# stick the following in a file called something like proxy.conf, so t
# you can include just for individual locations/sites.
#
gzip_proxied any;
gzip_vary on;
proxy_cache one;
proxy_cache_key $scheme$host$request_uri;
proxy_cache_revalidate on;
proxy_cache_valid 200 301 302 24h;
proxy_cache_valid 404 30m;
# proxy_cache_valid any 10m;
proxy_cache_use_stale error timeout invalid_header updating http_500 h
# proxy_connect_timeout 5s;
proxy_http_version 1.1;
proxy_redirect off;
# proxy_ignore_headers X-Accel-Expires Expires Cache-Control;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For  $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
```

In the above example, if the edge cache receives a file that doesn't have expires or cache-control headers, it will set 24 hours for anything with status codes of 200/301/302, and 30 minutes for any 404's as a catch-all. We've commented out the "any", so that any other status codes (like 500 errors) will attempt to be refetched and won't be normally be cached. Note that nginx will not use these if your source files have cache-control/expires headers (which ideally, they should) – it's only default behaviour for anything that doesn't.

> ASIDE: If we wanted to IGNORE what the origin says, we could UNCOMMENT the "proxy_ignore" line. Then nginx would ignore a number of those headers that specify caching and use the values we set here regardless – BE CAREFUL if you do that though – sometimes authenticated content relies on having "no cache" style headers, and you could be subverting that level of "security". Normally cookies will still take effect and disable the cache (unless you add cookies to that line which I do not recommend…), but still, you're better off setting up the caching via your origin's headers rather than forcing it here unless you have no other option.

Moving back up, the **proxy_cache_use_stale** gives circumstances where stale content can be served even if past the "deadline", the notion being that if your origin starts spazzing out, it's probably better to serve an old page than it is to serve an error to a visitor. Note that nginx does not currently support **stale-while-revalidate**, so you'll always have 1 unlucky visitor who has to wait for the proxy to revalidate when the time has run out.

```
# These are some WordPress settings, pulled from one of the plugins
 # (I don't remember which at the moment - probably W3TC or WP SuperCa
 # You can use them for WordPress or get a notion of what they do by r
 # through the code and modifying them for another CMS if necessary.
 #
 # You would only want it used for edges that serve WordPress sites,
 # though it shouldn't harm static websites. Attach
 # it to the end of the above proxy.conf example if using WordPress.
 #
 # If you're serving up some WordPress, some Joomla, etc, you may want
 # conf file and included it just for the WordPress sites.
 #
#Cache everything by default
 set $no_cache 0;
#Don't cache POST requests
 if ($request_method = POST)
 {
 set $no_cache 1;
 }
#Don't cache if the URL contains a query string
 #if ($query_string != "")
 #{
 #    set $no_cache 1;
 #}
# Don't cache uris containing the following segments
 if ($request_uri ~* "(/wp-admin/|/xmlrpc.php|/wp-(app|cron|login|regi
 set $no_cache 1;
 }
# Don't use the cache for logged in users or recent commenters
 if ($http_cookie ~* "comment_author|wordpress_[a-f0-9]+|wp-postpass|w
 set $no_cache 1;
 }
proxy_cache_bypass $no_cache;
 proxy_no_cache $no_cache;
 add_header X-MyProxyServerOne $upstream_cache_status;
```

The above (sorry about the mess) is meant for WordPress and essentially decides whether to cache on the edge cache or not based on the URL and specific cookies. You could modify it for other CMS's – the logic behind it will be similar. If you're using fastcgi caching on your origin you'll want to apply a slightly-modified version of the above there too. The final **add_header** line is for you – in it's current form it will add a header called **X-MyProxyServerOne** to all the pages it serves, and will attach hits/misses/etc. Customize the name and ensure that you are BYPASSING when logged in to WordPress or when you have left a comment etc. You can view the result by looking at the headers. If you use sites like WebPageTest or GTMetrix, you can view the headers there too, to see which of your edge servers served up the page. Just be sure to name each edge cache differently (do not name them all MyProxyServerOne).

```
#
 # This next bit is for if you're using SSL - you can toss it in your
 # in the conf file with your sites (probably not within a location bl
 # Read the nginx documentation to tune this.
 #
 ssl_session_cache    shared:SSL:2m;
 ssl_session_timeout  20m;
 ssl_ciphers ALL:!kEDH!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2
 # ssl_protocols SSLv3 TLSv1;
 ssl_prefer_server_ciphers on;
 ssl_session_ticket_key /etc/nginx/ssl-ticket.key;
```

Read up on those settings via the nginx documentation and tweak as needed. If you're not using SSL, you don't need it.

```
# The next part sets a lot of the buffering parameters. You should pro
 # details on the nginx site for details. In short though, we've told
 # 2GB of data in the cache (pushing out least-recently-used items if
 # not to physically evict anything that hasn't been accessed for 7 da
 # for this "high" 7 days is that if we can not reach the origin for s
 # nginx will be able to serve whatever-it-had in the cache even if it
 #(the conditions being based on our proxy_cache_use_stale settings me
 #
 # This doesn't go in a location block (put it above all your location
 proxy_buffering on;
 proxy_cache_path /etc/nginx/cache levels=1:2 keys_zone=one:40m inacti
 proxy_temp_path /etc/nginx/tmp;
 proxy_buffer_size 4k;
 proxy_buffers 100 8k;
 proxy_connect_timeout 7s;
 proxy_send_timeout 10s;
 proxy_read_timeout 10s;
```

You will want to tune the above based on your server configs. A low memory / HD server might need lower **keys_zone** and **max_size** values, while servers that aren't always going to be quick to respond might need longer timeouts, etc.

```
# The following backend items are fairly self-explanatory. You could p
 # the 1st item to the nearest edge cache which might be quicker if it
 # primed (or will prime it otherwise). Just make sure the neighbour
 # you are pointing to points to the origin, or you'll just create an
 # Make use of the "backup" if you go this route and point that at the
```

```
#
# Pointing to the origin keeps the complexity down and you won't acci
# create a loop. You probably want to remove the backup in that case
# do not accidentally create a loop if the primary goes down.
upstream backend {
server 11.11.111.111;
server 22.22.22.222   backup;
}
upstream backend-ssl {
server 11.11.111.111:443;
server 22.22.222.222:443   backup;
}
#
# example.com (non-SSL)
#
server {
listen 80;
listen [1a00:a110:1:1::1aa1:aaa1]:80;
server_name example.com;
location / {
proxy_pass http://backend;
include /etc/nginx/proxy.conf;
}
}
#
# example.com (SSL, requires that you've created CRT and KEY)
#
server {
listen 443 ssl spdy;
listen [1a11:a111:1:1::1aa1:aaa1]:443 ssl spdy;
server_name example.com;
ssl_certificate /etc/nginx/certificates/example-certificiate.crt;
ssl_certificate_key /etc/nginx/keys/example-key.key;
location / {
proxy_pass https://backend-ssl;
include /etc/nginx/proxy.conf;
}
}
```

The proxy.conf we **include** is essentially the proxy.conf file we mentioned earlier (you could integrate it without an include if you want). You can add more backend servers if you'd like – any that don't say "backup" will be used in a round-robin configuration. It is possible to assign weighting as well but you'll need to read the nginx documentation for that. You do need separate non-SSL and SSL locations if you want your edges to communicate with your origin via SSL based on the protocol your visitors used to reach the edge proxy. If that backend encryption isn't terribly important (or is VERY important), you could potentially merge them and have the backends communicate via all HTTPS or all HTTP.

Going back to your origin server, a lot of CMS's make use of incoming IP addressees. For example, WordPress comments tell you the IP of the visitor who left it. You probably don't want it to always show the IP address of your edge servers, so if your origin also runs nginx, you can pass along the real ones IPs like so:

```
#
 # Add to the nginx config on the ORIGIN server.
 #
 set_real_ip_from  11.11.111.111;
 set_real_ip_from  22.22.222.222;
 set_real_ip_from  33.33.333.333;
 set_real_ip_from  1111:1111:111:1::/64;
 set_real_ip_from  2a22:a222:2:2::/64;
 set_real_ip_from  3333:333::/32;
 real_ip_header    X-Forwarded-For;
 real_ip_recursive on;
```

Remember, this part above goes on your origin server – the one that's actually hosting/running your CMS's. Replace the IP addresses with the IP's of your edge caches, and nginx will "trust" them to provide the real IP. Note that the "recursive" part is quite important if you want nginx to verify the "chain of trust" here. What nginx will do is look through the "chain" of IP addresses and use the most "recent" one that you did not list above as the "real" one. Without it, it will just trust whatever IP address was fed as the original, which anybody could spoof rather easily. As a note, the bottom 3 IP addresses are various methods of displaying the original IPv6 ranges. Because IP6 addresses are often distributed in various ways (you may have been assigned a chunk of which your server randomly uses one, or you could have 1 specific IP address), you can plunk your IPv6 address into various tools on the web and it'll generate the proper 32/64/etc if needed.

If your origin does not use nginx, there's probably an equivalent way to do it in Apache. Assuming you're using WordPress (or another CMS), there are numerous guides on the web that will let you pull the "real" ip address via PHP instead, though note that some methods aren't extensive and you'll just get whatever IP address your visitor claimed to be coming from in that case (in other words, some methods are easier for someone to spoof so bear that in mind if spoofing would pose a security problem for your site).

Ok, that was long. Get a coffee. You deserve it! Then come back and we'll continue.

Testing your first edge-cache

As excited as you might be to launch everything (now that nginx hopefully restarted), you need to test first.

First, restart the edge server. Make sure it does start, and that nginx comes back up (chances are you'll find out by visiting the ip address directly, hitting SHIFT-refresh and making sure the default page pops up). If nginx won't autostart when the server is rebooted and you're on a KVM VPS and are using IPv6, it's possible you're running into this issue.

Next, you want to test the behaviour of your sites through the edge. The easiest way to do this is to edit your local HOSTS file (on the computer you're using right now!). If you've just configured nginx and don't know how to edit a hosts file, (1) I'm incredibly impressed, (2) look up where it is on your OS and add a line containing the IP address of your edge server and name of your site as so:

```
11.11.111.111 example.com
```

Save it, and then open a new web browser (one that was already open might have cached your old hosts file), and then try to visit the site. Assuming it comes up and looks good, check the headers by using “inspect element” in Chrome/Firefox/etc, find the response/headers portion, and make sure that your special X-MyProxyServerOne header is showing up with the other headers – if it doesn’t, your browser might still be visiting the old site in which case you’ll have to play a bit (worst-case, usually restarting your computer will force everything to look at the new hosts file).

Try various parts of your site (pages, images, css, javascript, favicon, etc) and **watch the headers**. Make sure expires headers look good. Make sure that when you’ve authenticated (leaving a comment in WordPress or logging into the admin section for example), you’re getting BYPASS messages. Note that once you’re at that point where cookies are triggering the bypass, you’ll usually have to clear your cache/cookies/etc before you can see what a “normal” user does again. Open another web browser simultaneously (or a “Private” mode) and check again to make sure unauthenticated users will not be seeing your cached admin pages or admin access.

Spend a lot of time testing, and do any tuning now, because you’ll probably be copy/pasting your first config to your additional edge servers and it’s easier to do it all now – from here on, any tweak you make after-the-fact will have to applied to each edge server.

Additional servers

Once it all checks out, you can add any additional tuning you may want and then pretty much copy/paste your nginx config (and any other server config you did) to additional servers to use as edges, making sure to update the IP addresses for each location. You should still test those ones, but you shouldn't have to be as thorough assuming that they're virtual mirrors of your original edge cache (same OS, same nginx install, same settings elsewhere).

One thing you will want to decide is the way you'll "flow" information to the edge caches. As I alluded to in the first image, one way is to have every edge pull from the origin. It's simple, and it works. However, if your sites aren't very busy, or if you're using low expiration times, you could consider daisy-chaining your edge caches to keep them all populated. For example:

ORIGIN (western US) -> **edge 1** (eastern US) -> **edge 2** (Europe) -> **edge 3** (China) -> **edge 4** (Australia)

Let's look at that from the perspective of a visitor in Europe. Chances are that if "their" edge server requested from the origin in the western US, it would travel all the way across the US before crossing the ocean and getting to them. So if your eastern US edge server already has the resource they requested, it makes sense to grab it from there since it's a shorter distance, and should be lower latency. Of course, if the eastern US one doesn't that page cached already, it'll add a little time to the request – probably not a whole lot since it's "on the way", but it'll add some overhead regardless. The plus side is that the eastern server now has a fresh copy too, so the time that Mr Europe lost will be gained by Mr EastCoaster's next visit.

This only goes so far though. Look at the guy in Australia. If for some reason the item he requested isn't on his Australia "edge", or in China, or in Europe, he's basically waiting for the resource to make it's way around the globe when it could have just made a short trip across the ocean. Then again, his wait might have saved the next europe & china visitors some time.

So consider daisy-chaining, but keep in mind the potential for 1 person at the end of the chain to get really, really, screwed. Also remember the previous note about "cache control" vs "expires" – in this case, with 24 hour "cache control" times, the guy in Australia might see content that's almost 100 hours old. Expires might make more sense here, or perhaps limiting "chains" to 2 hops in each direction. Note that if your backends communicate with each other over SSL, each hop will actually add a non-insignificant amount of time even if it's on the way.

Part 3 – The DNS

I'll assume you finished that and came back from another coffee break.

Now that it's all functional, it's time to set up the DNS. I'll look at the Amazon Route 53 and Rage4 DNS stuff here.

For Route 53… actually… I hope you read this far before you ran out and bought some VPS's…. In any case, Route 53 makes use of "Regions", which coincide with various Amazon datacenter locations, so you're looking at a MAX of 8 possible latency-based locations, and you'll want to make sure that the VPS's you get match the location of Amazon's datacenters. Because it's always a bear to find (Amazon hides it in shame), here's the current list:

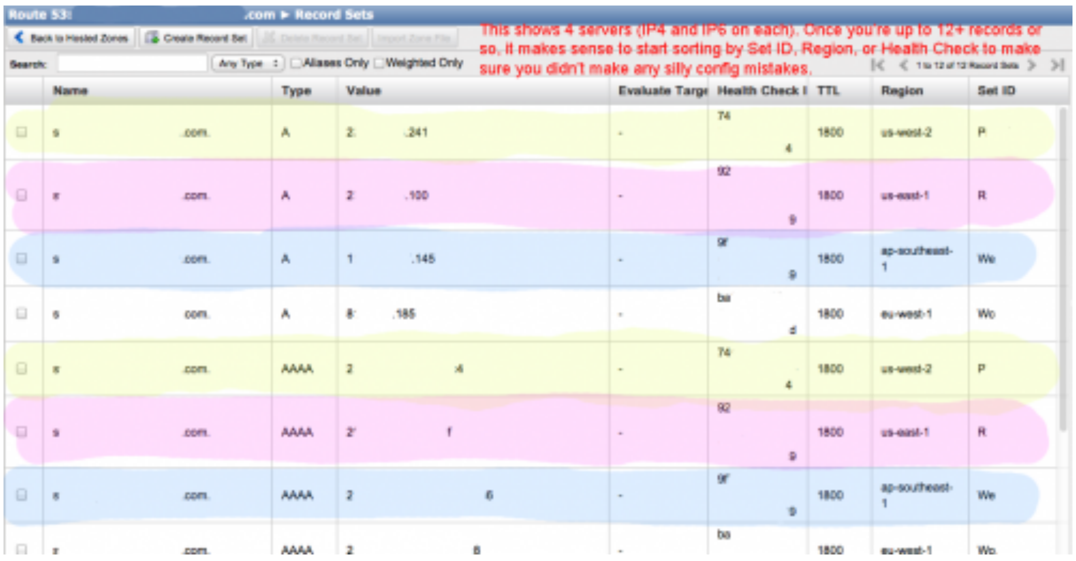
us-east-1 – US East – Northern Virginia
us-west-1 – US West 1 – Northern California
us-west-2 – US West 2 – Oregon
eu-west-1 – Europe – Ireland
ap-southeast-1 – Asia Pacific Southeast 1 – Singapore
ap-southeast-2 – Asia Pacific Southeast 2 – Sydney
ap-northeast-1 – Asia Pacific Northeast 1 – Tokyo
sa-east-1 – South America – Sao Paulo

So Amazon isn't great if you wanted a north/south distribution in the US, or 4 geographically diverse servers within Europe. No, you have to pretty much follow those guidelines because that's how your traffic will be targeted. The good news is that Amazon's routing is pretty darn accurate.

Important to note that you don't need to set servers for each of those – in other words, you do not need 8 servers. You could only use 2 servers (say, in Western US and Europe), and as long as you picked **us-west-1** and **eu-west-1**, traffic around the world would generally be routed to the server that gives the lowest latency based on their location – Amazon has this flexibility automatically configured and it works well.

To set it up, Amazon gives you 4 DNS servers to use (a .com, .net, .org, and .co.uk), and you create each A/AAAA record manually. You would typically use "Latency Based Routing" in the setup we're targeting. The setup is fairly easy (for being Amazon, anyway) and their site is always responsive so it's pretty quick once you get a workflow going. Things can get a little hairy when you're copy/pasting a bunch of IPv6 addresses in for a bunch of subdomains, but the UI is fairly decent all things considered. You'll pay the $0.50 for the domain within a day, and you'll see your first penny charged for queries fairly quickly.

Here are a few screenshots from Amazon Route 53, configured in such a way that you've got 4 servers, each with IPv4 and IPv6 AAAA addresses that you're using (so 8 records for just the .website.com, and an additionall 8 for each subdomain). The "physical" servers are color-coded here:



Route 53: .com ▸ Record Sets

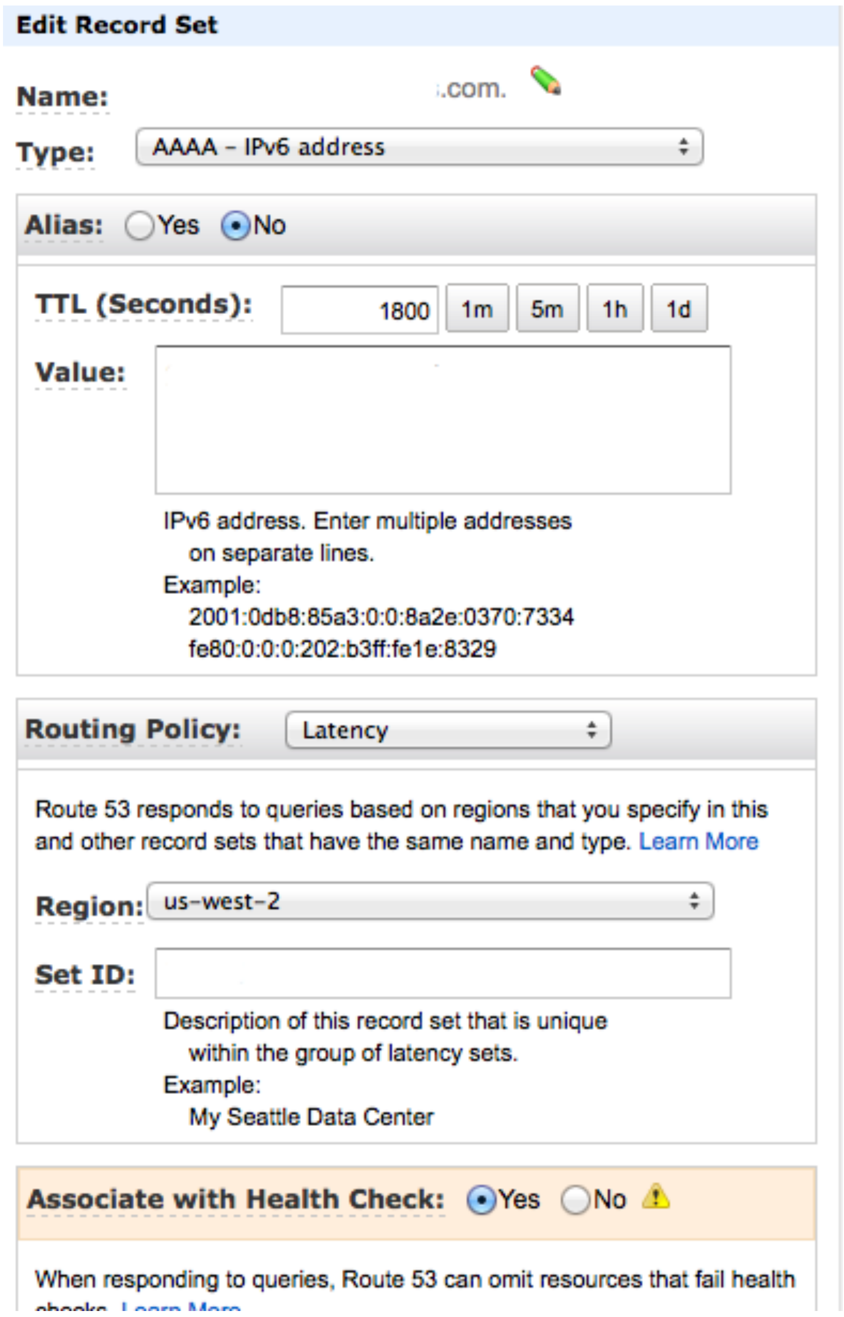
Back to Hosted Zones | Create Record Set | Delete Record Set | Import Zone File

Search: Any Type | Aliases Only | Weighted Only

This shows 4 servers (IP4 and IP6 on each). Once you're up to 12+ records or so, it makes sense to start sorting by Set ID, Region, or Health Check to make sure you didn't make any silly config mistakes.

1 to 12 of 12 Record Sets

	Name	Type	Value	Evaluate Targe	Health Check I	TTL	Region	Set ID
☐	s .com.	A	2: .241	-	74 4	1800	us-west-2	P
☐	r .com.	A	2 .100	-	92 9	1800	us-east-1	R
☐	s .com.	A	1 .145	-	9f 9	1800	ap-southeast-1	We
☐	s .com.	A	8 .185	-	ba d	1800	eu-west-1	Wo
☐	r .com.	AAAA	2 :4	-	74 4	1800	us-west-2	P
☐	s .com.	AAAA	2' f	-	92 9	1800	us-east-1	R
☐	s .com.	AAAA	2 6	-	9f 9	1800	ap-southeast-1	We
☐	r .com.	AAAA	2 8	-	ba	1800	eu-west-1	Wo



Edit Record Set

Name: .com.

Type: AAAA – IPv6 address

Alias: Yes [✓ No]

TTL (Seconds): 1800 | 1m | 5m | 1h | 1d

Value:

IPv6 address. Enter multiple addresses
on separate lines.
Example:
2001:0db8:85a3:0:0:8a2e:0370:7334
fe80:0:0:0:202:b3ff:fe1e:8329

Routing Policy: Latency

Route 53 responds to queries based on regions that you specify in this and other record sets that have the same name and type. Learn More

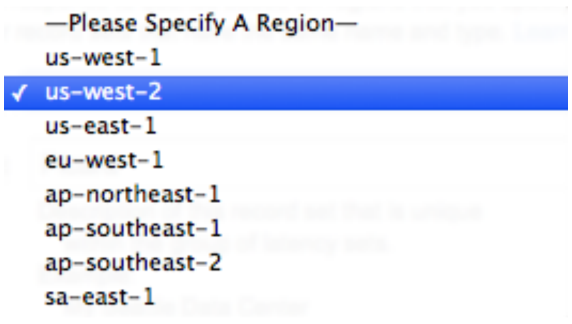
Region: us-west-2

Set ID:

Description of this record set that is unique
within the group of latency sets.
Example:
My Seattle Data Center

Associate with Health Check: [✓ Yes] No ⚠

When responding to queries, Route 53 can omit resources that fail health checks. Learn More



Optionally, you can set up “Health Checks” as well ($0.75/month each), and tie each DNS record to a health check. If you have 4 servers, you might have 4 health checks going to monitor each one ($3/month total), using the IP address and ideally a host name, HTTP or TCP port, and an optional path to a file. If that becomes non-responsive or spits out an HTTP status code that isn’t within the 200-399 range, Amazon will “remove” that server from the pool until it’s back up – traffic will be distributed via DNS as though that server doesn’t exist. Incidentally, responsive means that Amazon connects within 4 seconds, and then receives the proper HTTP code within 2 seconds. A huge advantage here is that you can apply those health checks to as many A/AAAA records as you want. So if you’re hosting 100 websites on 2 servers (each server with a health check), you can apply those 2 health checks to all 100 websites if you want. Not bad for $1.50.

If you go with the health checks, note that Amazon won’t alert you when the site is down unless you set up the extra Amazon alarms which cost extra and are a whole other section in AWS (you can visit the health-check history graph in the Route53 interface though). So keep in mind that if you don’t want to set up alarms, you’ll want to set up an alternate method for finding out if your server has crashed. Free uptime monitors tend to vary in reliability (many false positives), but may suffice.

Rage4

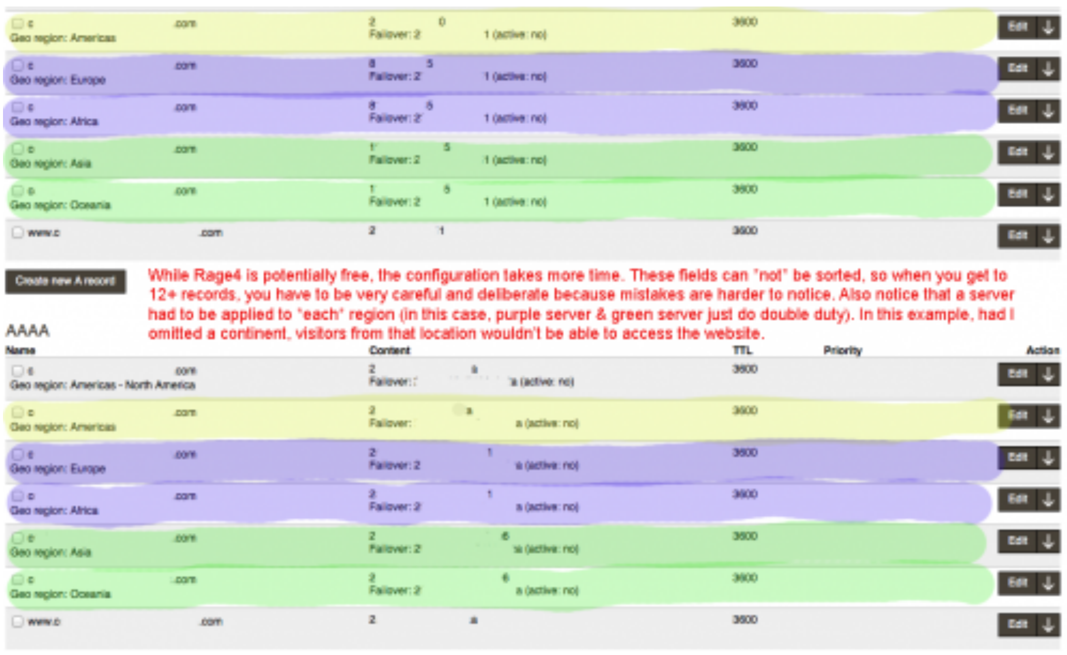
For a low-traffic site, it’s hard to beat free. And unlike Amazon, Rage4’s DNS servers are dual-stack, meaning you can end up with fully IPv6-accessible site. If you plan to have many servers in many locations, where

Amazon offers 8 regions, Rage4 offers a whopping 34 with the US being broken into 10, Europe into 4 (NESW), Canada into 2 (E/W), and others quite a bit more fine-tuned than Amazon.

To use that GeoDNS region-based targetting, you can target at different region-levels. Note that a big downside to Rage4 is that if you misconfigure something (like forgetting a continent), you'll have a lot of visitors that will not reach your site. "Global Availability" makes an IP available everywhere. You probably won't want that, so for starters you might assign a server to each continent. If you have 2 servers total (1 US server and 1 EU server), you might set the US one to cover the Americas, China, and Oceania (3 records), and set the EU one to cover Europe and South America (2 records).

With the region-based targeting, things start to get a little hairy after that. What if you have 2 US servers – one on the east coast, one on the west cost? You'll probably want to set up all 10 US-based regions, and the east/west Canada ones. That's 12 records you'll have to create. And then you still have to create records for all the other regions (Mexico, Central/South America, Europe, Asia, etc). It gets pretty complicated pretty fast.

Here's a screenshot showing how things might look if you're doing something simple like using 3 servers (1 in the US, 1 in Europe, 1 in Oceana).



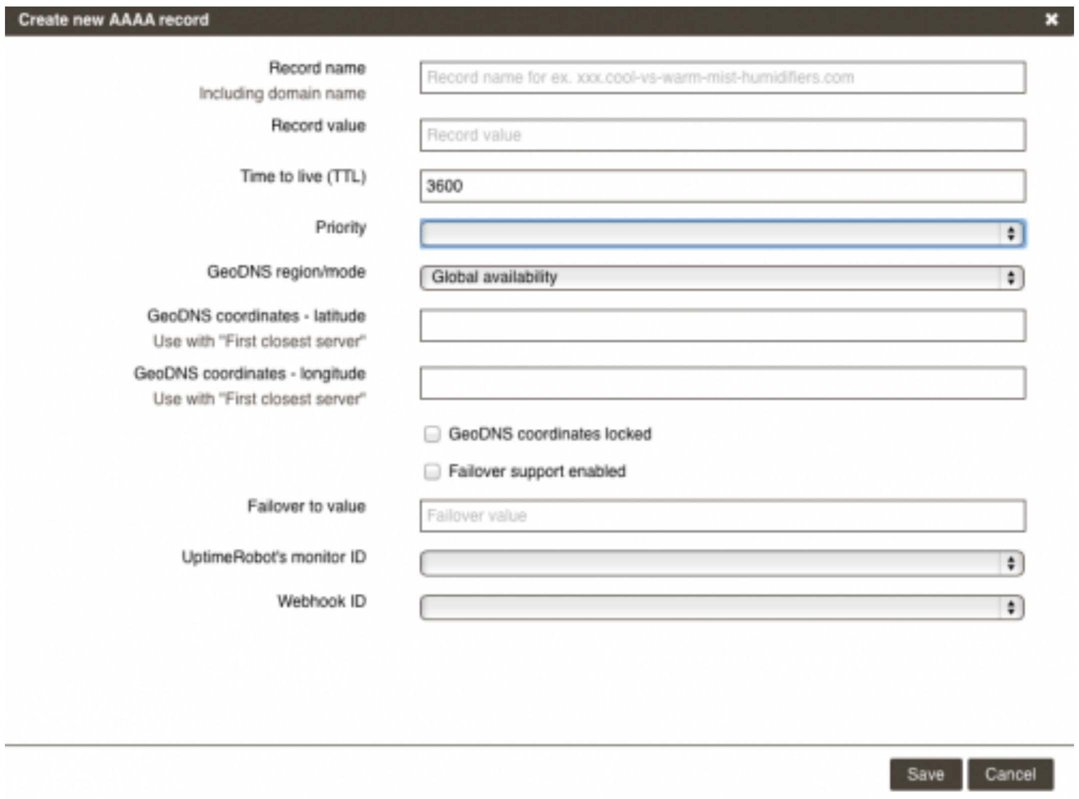
Name	Content	TTL	Priority	Action
c .com Geo region: Americas	2 0 Failover: 2 1 (active: no)	3600		Edit
c .com Geo region: Europe	8 5 Failover: 2 1 (active: no)	3600		Edit
c .com Geo region: Africa	8 5 Failover: 2 1 (active: no)	3600		Edit
c .com Geo region: Asia	1 5 Failover: 2 1 (active: no)	3600		Edit
c .com Geo region: Oceania	1 5 Failover: 2 1 (active: no)	3600		Edit
www.c .com	2 1	3600		Edit

Create new A record

While Rage4 is potentially free, the configuration takes more time. These fields can "not" be sorted, so when you get to 12+ records, you have to be very careful and deliberate because mistakes are harder to notice. Also notice that a server had to be applied to "each" region (in this case, purple server & green server just do double duty). In this example, had I omitted a continent, visitors from that location wouldn't be able to access the website.

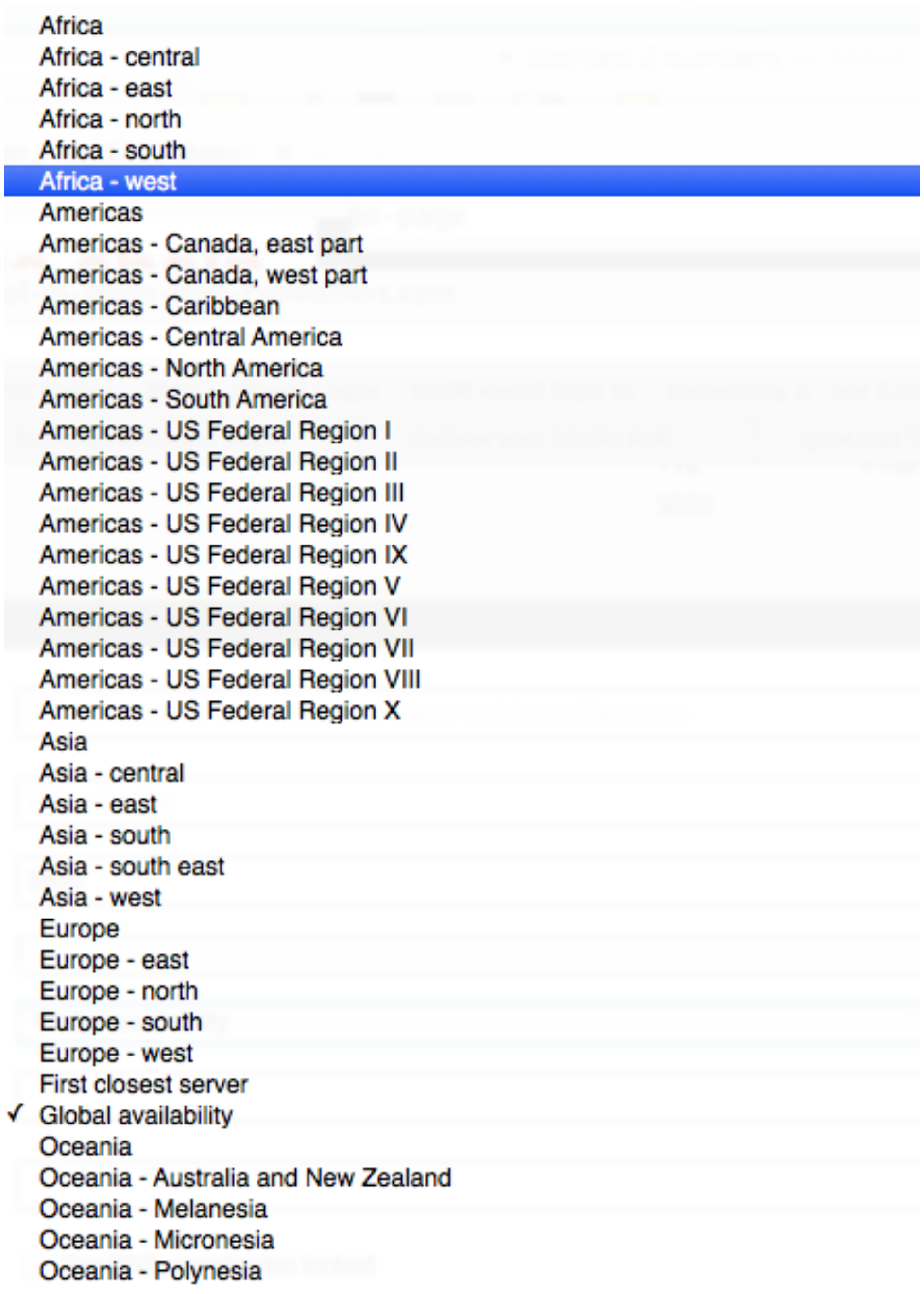
AAAA

Name	Content	TTL	Priority	Action
c .com Geo region: Americas - North America	2 a Failover: a (active: no)	3600		Edit
c .com Geo region: Americas	2 a Failover: a (active: no)	3600		Edit
c .com Geo region: Europe	2 1 Failover: 2 a (active: no)	3600		Edit
c .com Geo region: Africa	2 1 Failover: 2 a (active: no)	3600		Edit
c .com Geo region: Asia	2 6 Failover: 2 a (active: no)	3600		Edit
c .com Geo region: Oceania	2 6 Failover: 2 a (active: no)	3600		Edit
www.c .com	2 a	3600		Edit



Create new AAAA record

Record name Including domain name	Record name for ex. xxx.cool-vs-warm-mist-humidifiers.com
Record value	Record value
Time to live (TTL)	3600
Priority	
GeoDNS region/mode	Global availability
GeoDNS coordinates - latitude Use with "First closest server"	
GeoDNS coordinates - longitude Use with "First closest server"	
	☐ GeoDNS coordinates locked
	☐ Failover support enabled
Failover to value	Failover value
UptimeRobot's monitor ID	
Webhook ID	


Africa
Africa - central
Africa - east
Africa - north
Africa - south
Africa - west
Americas
Americas - Canada, east part
Americas - Canada, west part
Americas - Caribbean
Americas - Central America
Americas - North America
Americas - South America
Americas - US Federal Region I
Americas - US Federal Region II
Americas - US Federal Region III
Americas - US Federal Region IV
Americas - US Federal Region IX
Americas - US Federal Region V
Americas - US Federal Region VI
Americas - US Federal Region VII
Americas - US Federal Region VIII
Americas - US Federal Region X
Asia
Asia - central
Asia - east
Asia - south
Asia - south east
Asia - west
Europe
Europe - east
Europe - north
Europe - south
Europe - west
First closest server
✓ Global availability
Oceania
Oceania - Australia and New Zealand
Oceania - Melanesia
Oceania - Micronesia
Oceania - Polynesia

So while Rage4 gives the flexibility of 34 GeoDNS regions, you might start to get a little overwhelmed when it feels like you're forced to set up a lot of locations one at a time just because you have 2 US servers. And that's just for one A record. You might have to do the same for your AAAA records. And if that was for your root domain, you might have to do the same for www. It gets crazy. The UI isn't as responsive as Amazon's by any means, so that can add to the frustration. Note that if you set a server as "Global Availability", one as "Americas", one as "North America", and one as a specific "Canada East", anybody from eastern Canada who tries to visit your site will be returned ALL 5 records. So they'll randomly hit one of those servers, which is usually not what you want.

Alternately they have a "First Closest Server" option, where from what I understand you plunk in the co-ordinates of your servers, and it'll compare your visitor's co-ordinates with your servers to find the closest one geographically. I say "from what I understand", because I couldn't figure out how to make this option work. It asks for latitude and longitude – I looked up my server's cities in Google and got them. But it didn't seem to accept N or W, and there wasn't any indication as to how North or South were differentiated.

There's probably a simple answer to the "first closest server" issue, but I couldn't find it in the documentation. It seems as though the service may be targeted more towards web hosts who have an underlying system as to how the GeoIP feature they use works, which is a shame because the service really offers a lot that could benefit a lot of people – it just needs a little more ease of use. It's too easy to configure something incorrectly and have traffic in a certain part of the world that can't reach your site, or traffic from a certain region that gets routed to servers you didn't intend. If you use Rage4, I strongly recommend running Website Tests from various locations around the globe to make sure everything is accessible.

The failover feature is handled by linking to your UptimeRobot monitor. UptimeRobot's a free service, and it throws a lot of false positives if you have it look for a keyword in the content (fewer false positives for more simple monitoring), but for free it will monitor a *lot* of websites, and you can tie these to the Rage4 failover feature via an API key. You then enter an IP address for your record to fail over to, and that's that.

So… Route53 vs Rage4…?

Speed-wise, I ran a number of tests and both traded wins to the point where I wouldn't be comfortable stating a winner there. For Geo-accuracy, if tools like super-ping are to be believed (a stretch, since web servers generally aren't the focus of GeoIP location), Amazon is definitely closer to the mark there.

Rage4 offers IPv6-accessible DNS servers, free monitoring/failover (via UptimeRobot at 5min intervals), and more finely-aimed Geo-targetting. If you're under the 250,000 queries/month, it's essentially free for the respective domains. It becomes more expensive than Amazon once you exceed that number. If you have a few domains, assume $2.75 USD for each domain that crosses that threshold (so long as it's still under 1 mil) with the others being free. It is more complex and time consuming to set up (I shudder to consider how long it would take to set up a site with 10 subdomains across 5 servers with both IP4/6 addresses…). But if you're running a lot of servers (multiples in each country), it might be your best option.

Amazon Route 53 is much easier to set up, and pricing is nice and low, but you're limited when it comes to region targeting. If you're willing to limit yourself to servers near the regions they mention though, it's very hands-free and hard to get wrong (typos excluded of course). Taking a 2nd look over your records for errors is easier either way. Beyond that, assign each IP to a region, and Amazon takes it from there. Failover monitoring gets pricey as you add multiple servers, but is much quicker to detect failures (30 second intervals), very reliable, and because it simply pulls failed servers from the pool, you don't have to worry that it pulled 2 servers that may have listed each other as fallbacks.

As a recommendation…. if IP6 at the DNS level is important to you or you're going to have multiple servers per region and don't mind the painful process of getting it configured, Rage4 is probably the one you want to look at. If smooth & reliable uptime monitoring/failover behavior is key, or you know you'll be hitting millions of queries and are price-conscious, Amazon is the way to go. If you're not sure either way, I'd suggest starting a small site with Rage4. If you can figure it all out and everything works great, you may have found your match. If you're lost and confused partway through, give Amazon a shot.

Part 4 – We're done!

Hopefully the time you spent reading here saves you hours when you actually go to set everything up. Remember to test frequently along the way. When you first transition sites to use your new "CDN' or "edge cache", you might want to set every type of "expires" to low values in case something goes wrong. This goes for the DNS stuff too – use low TTL values. There's nothing worse than having something incorrectly cached somewhere for days, months, or years! Once you're sure everything's working right, bump up the times gradually.

If you need clarification on any of the steps, found that I've said something utterly wrong/boneheaded, or have tips that might help out others (perhaps even a totally different way of doing things!), feel free to leave a comment!