# Chapter 10. DNS with Route 53

Amazon’s Route 53 service can provide a tighter integration between DNS and other AWS systems such as Elastic Load Balancers and the Elastic Compute Cloud. The venerable Domain Name System remains a critical component in the knowledge toolbox of a system administrator. Although DNS is incredibly simple at its core, a broken or misconfigured DNS server can result in some interesting problems. Kris Buytaert, one of the original proponents of the DevOps movement, notes this in the title of his blog: [Everything Is a Freaking DNS Problem](http://www.krisbuytaert.be/blog/).

This book assumes that most readers are familiar with the general concepts surrounding DNS. This chapter therefore focuses on the AWS-specific implementation provided by Route 53, and demonstrates a few techniques that can be used to configure a cloud-aware DNS service.

# Why Use Route 53?

As mentioned in previous chapters, many AWS services are similar to their non-cloud counterparts. Route 53 continues this theme. It is possible to use Route 53 as a replacement for a traditional web-based DNS service, and indeed many people are using Route 53 in exactly this way. However, some features specific to Route 53 make it the ideal method of managing DNS when operating or managing an AWS infrastructure.

The main reason for this is that it is tightly integrated with services like Elastic Load Balancers. Attempting to use an external DNS service in combination with AWS can necessitate a large amount of custom development, such as writing scripts to query the status and hostname of an Elastic Load Balancer and continuously updating your DNS records to reflect changes to the ELB.

Route 53 provides this feature through the Alias resource record type. This is an Amazon-specific record type and should not be confused with a CNAME record type, which can also be used to alias one hostname to another. In Amazon terms, an Alias resource record type allows a hostname to be linked to a specific AWS service endpoint, such as the hostname of an ELB or a website hosted in an S3 bucket.

Other Amazon-specific features include Latency Based Routing, whereby the result of a DNS query can change depending on the location of the end user and other factors such as network link speed. The idea is to route the user’s request to the service that can provide the best experience to the user. For example, one might be serving a web application from two AWS regions, such as Sydney and Ireland. Using Latency Based Routing, you can ensure that requests from London-based users are routed to the web application servers in Ireland, which will provide much better performance than the Sydney-based servers. However good Amazon’s service is, the speed of light remains a hard limit to the speed with which one can serve up web applications to a geographically diverse user base.

From an operational perspective, Route 53 is just another AWS service, meaning it can be managed from the Management Console, command-line tools, or even CloudFormation. One common complaint about DNS services that provide only a web interface is that they allow no way to back up and restore your resource records. Many sysadmins also prefer a command-line option for managing their DNS records, or perhaps an API. Route 53 can provide all of these options, making it a good choice for DNS management even if you are not using many other AWS systems.

# Failure Is an Option: Using Route 53 to Handle Service Failover

The first scenario we will look at is using DNS to manage service failure. As much as we might wish otherwise, services will break from time to time, and we need to ensure that the interruption is as brief as possible from the user’s perspective.

Consider a PostgreSQL database cluster that consists of a master (which can handle both read and write traffic) and a slave (which can handle only read traffic). In this common scenario, a high level of uptime is required, or the single master is not capable of handling all of the application load on its own. Using tools such as [repmgr](http://www.repmgr.org/), you can easily configure a PostgreSQL cluster consisting of a single master and multiple slaves. Furthermore, repmgr can be used to periodically check the health of the master server and automatically promote one of the slaves in the event of a failure.

Many applications can be configured to send read traffic to one address, while read/write traffic is sent to another address. This makes scaling up the application traffic much easier, because you can offload read-only traffic from the master to the slave. For example, all read-only traffic would be sent to slave.example.com, while read/write traffic is sent to master.example.com.

The slave.example.com DNS record can be configured to return multiple addresses in response to client queries. With this configuration, your application can send all traffic to a single hostname without being aware of how many PostgreSQL slaves are currently in service.

With this in mind, consider how the dynamic nature of Route 53 can be used in combination with PostgreSQL (or indeed, many other services) failover.

In the initial state where everything is working correctly, the master.example.com record points to the working master. Repmgr exposes hooks allowing users to run custom scripts when a failover or promotion event occurs. Using these hooks, it is possible to automatically update the master.example.com DNS record to point to the newly promoted master server, which was previously in operation as a slave. It is also necessary to remove the corresponding entry from the slave.example.com record pool.

When using this method, it is important to consider DNS caching and how it will affect failover time. The DNS record’s TTL (time to live) setting indicates how long the result should be cached by clients, but not all DNS clients honor this setting. Some (arguably broken) applications cache DNS settings permanently and will recognize the updated DNS records only after being restarted.

This method can be implemented in various ways, depending on your infrastructure and the services you are running. To make this example more practical, we will implement the features just described using a PostgreSQL cluster with streaming replication and failover.

Configuring the PostgreSQL cluster is beyond the scope of this book, and is superbly documented in the official PostgreSQL documentation. Using this in combination with [repmgr failover documentation](https://github.com/2ndQuadrant/repmgr/blob/REL2_0_STABLE/autofailover_quick_setup.rst), begin by configuring a PostgreSQL cluster with a master and at least one slave. The master and slave(s) should be running on separate EC2 instances.

For the sake of the demonstration, it is suitable to launch individual instances for each role. In production, the instances likely will be running in Auto Scaling groups so that instances are automatically replaced if they fail.

As each PostgreSQL instance is launched, it must register itself with Route 53 by creating a CNAME record pointing to its hostname. This is done by running a script at launch time that creates the relevant DNS record. An example of such a script is shown in [Example 10-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch10.html#postgresql_launch_script).

##### ***Example 10-1. PostgreSQL launch script***

#!/usr/bin/python

import argparse

import boto.route53

from boto.utils import get\_instance\_metadata

def do\_startup():

""" This function is executed when the instance launches. The instances

IP will be added to the master or slave DNS record. If the record

does not exist it will be created.

"""

# Check if the master resource record exists

if zone.get\_cname(master\_hostname) is None:

print 'Creating master record: %s' % master\_hostname

status = zone.add\_cname(master\_hostname, instance\_ip, ttl)

return

print "Master record exists. Assuming slave role"

# Check if the slave resource record exists - if more than one result is found by get\_cname,

# an exception is raised. This means that more than one record exists so we can ignore it.

try:

slave\_rr\_exists = (zone.get\_cname(slave\_hostname) != None)

except boto.exception.TooManyRecordsException:

slave\_rr\_exists = True

if slave\_rr\_exists:

print 'Slave record exists. Adding instance to pool: %s' % slave\_hostname

else:

print 'Creating slave record: %s' % slave\_hostname

# Create or update the slave Weighted Resource Record Set

status = zone.add\_cname(slave\_hostname, instance\_ip, ttl, slave\_identifier)

def do\_promote():

master\_rr = zone.get\_cname(master\_hostname)

print 'Updating master record: %s %s' % (master\_hostname, instance\_ip)

zone.update\_cname(master\_hostname, instance\_ip)

# Remove this instance from the slave CNAME pool by deleting its WRRS

print 'Removing slave CNAME: %s %s' % (slave\_hostname, slave\_identifier)

zone.delete\_cname(slave\_hostname, slave\_identifier)

parser = argparse.ArgumentParser(description='Update Route 53 master/slave DNS records')

parser.add\_argument('action', choices=['startup', 'promote'])

#parser.add\_argument('--hosted-zone-id', required=True)

parser.add\_argument('--domain', required=True)

parser.add\_argument('--cluster-name', required=True)

parser.add\_argument('--test')

args = parser.parse\_args()

metadata = get\_instance\_metadata()

instance\_ip = metadata['local-ipv4']

instance\_id = metadata['instance-id']

ttl = 60 # seconds

master\_hostname = 'master-%s.%s' % (args.cluster\_name, args.domain)

slave\_hostname = 'slave-%s.%s' % (args.cluster\_name, args.domain)

# Identifier used for slave Weighted Resource Record Set

slave\_identifier = ('slave-%s' % instance\_id, 10)

conn = boto.route53.connect\_to\_region('eu-west-1')

zone = conn.get\_zone(args.domain)

if args.action == 'startup':

do\_startup()

elif args.action == 'promote':

do\_promote()

Execute this script manually on each of the instances, making sure to run it on the master first. You will also need to provide the hosted zone ID of your Route 53 zone and set theAWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY environment variables, for example:

export AWS\_ACCESS\_KEY\_ID=xxx

export AWS\_SECRET\_ACCESS\_KEY=yyy

python update\_route53.py --domain example.com --cluster-name db startup

After executing the script on both instances, you should see two new records in the Route 53 web console.

###### NOTE

This script is simple and requires a few tweaks to make it robust enough for production. For example, what happens if the master DNS record already exists, but the PostgreSQL service has failed? Should the script forcefully “take” the hostname and point it to the instance on which it is running? Automatic database failover requires plenty of careful thought before implementation.

Each node in the cluster will have its own repmgr.conf file, usually located at /etc/repmgr/repmgr.conf. This contains a promote\_command parameter that specifies the path to a script that will be executed when a failover event occurs. This is the hook we will use to update the DNS records when a slave is promoted to the master role.

The script in [Example 10-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch10.html#repmgr_promote_script) will be executed on the slave that is being promoted. Note that this script is also responsible for initiating the repmgr failover process.

##### ***Example 10-2. RepMgr promote script***

#/bin/bash

# Use repmgr to initiate the failover process

repmgr promote

# Run the script to update the DNS records

python update\_route53.py --domain example.com --cluster-name db promote

Save the script on both the master and slave instances, perhaps in /etc/repmgr/promote\_script.py, and make it executable. Update the repmgr.conf file so that the PROMOTE\_COMMAND parameter points to the path of the script.

With the DNS records created and the promote script in place, you can test the failover process. You might want to refer to the repmgr documentation again at this point for more details.

Stop the PostgreSQL service on the master instance and watch the repmgr log file. Once the repmgr daemon notices that the master has failed, the slave will be promoted by calling the script.

Once the process has completed, check the Route 53 Management Console. You will see that the master.example.com record now points to the instance that was previously a slave, and the corresponding slave.example.com record has been deleted.

When the PostgreSQL service was stopped on the master, any clients attempting to connect to it would have begun generating error messages. As the DNS change propagates to the clients, they will begin connecting to the new master and begin functioning correctly.

# Ramping Up Traffic

In the process described in the preceding section, traffic was abruptly shunted from one instance to another by changing the hostname. The instance is thrown in at the deep end and must immediately handle all traffic destined for its hostname, which could overload the instances, as they have not had time to warm up their caches. Sometimes, it is desirable to send traffic to an instance in a more controlled and gradual fashion. This is often true in the case of database services.

To continue the PostgreSQL example: A highly efficient PostgreSQL server relies on data being stored in memory and not read from spinning disks. Immediately sending a large number of queries to a recently started PostgreSQL instance will result in much of the data being read from disks, resulting in poor performance until PostgreSQL has had a chance to warm up the cache by storing recently used data and indexes in memory.

To work around this, we can begin by sending a small amount of traffic to a new PostgreSQL instance and gradually increasing this amount over time. For example, we could start by sending 10% of traffic to the instance and increasing this by 5% every five minutes. This can be done using [weighted resource record sets](http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/WeightedResourceRecordSets.html) (WRRS), which are used to return multiple records in response to a query for a single DNS hostname.

###### NOTE

Remember that DNS caching affects how quickly your application responds to changes in the resource record set.

When creating an entry (a member of the pool) in a WRRS, a weight value must be provided. This is used to calculate how frequently this record will be returned in response to client queries. For example, the slave.example.com hostname used for our slave database in the previous example could be configured to return multiple records, allowing traffic to be distributed across multiple slaves. If the records all have the same weight, traffic will be distributed in a round-robin fashion.

This method can also be used to perform a phased rollout of software updates. For example, a small percentage of web application traffic can be sent to instances running a new version of the software, while the majority of traffic is sent to instances running the existing stable version.

Once the new version is confirmed to be working as expected, the rest of the traffic can be shifted over to the new instances, or alternatively the other instances could be updated in situ.

# Surviving ELB and Application Outages with Route 53

Elastic Load Balancers are reliable, but like any component, they can sometimes experience failures. A robust infrastructure will take this into consideration and embody a way to work around temporarily unavailable ELBs. The default failure condition of a nonresponsive ELB does not make for a good user experience: users will see a blank, unstyled error page.

Outside AWS, one common method of working around failures is to have a separate web server that is responsible for serving your error page, or a message informing the user that the site is currently down for maintenance (scheduled or otherwise).

Within AWS, this process can be automated so that Route 53 will begin returning the IP address of your maintenance server instead of the IP address of your ELB. Furthermore, AWS can be used to serve your error pages from S3 buckets, removing the requirement of running an additional server solely for the purpose of serving error pages.

This is achieved using failover record sets. These work in the same way as weighted resource record sets with one important change: Route 53 will periodically perform health checks against your ELB (or other specified health check endpoint). Route 53 will respond to queries based on the results of this health check. For example, when your ELB is working normally, Route 53 will provide the ELB’s IP addresses in response to DNS queries. If the ELB (or the application behind it) fails these health checks, Route 53 will provide a different result to DNS queries.

In this example, we will use this feature to set up an S3 bucket capable of serving error pages, which will be used as a failover in case the ELB or application fails. Without any action on our part, Route 53 will automatically route users to this error page in the event of an ELB or application failure.

The first step is to create a bucket that will serve our error page. Using the S3 Management Console, create a new bucket and assign it a name like my-error-page. Follow the instructions in Amazon’s [Configure a Bucket for Website Hosting](http://docs.aws.amazon.com/AmazonS3/latest/dev/HowDoIWebsiteConfiguration.html) documentation to configure this bucket to serve web pages. When configuring the bucket to serve web pages, you will need to provide the name of the index document and error document. Set both of these to index.html to ensure that any requests that reach this bucket will result in your error page being served.

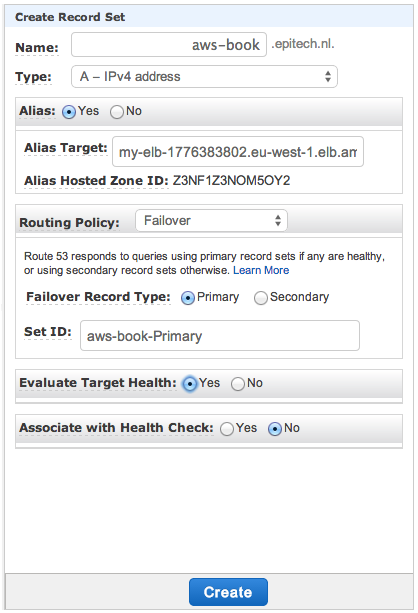
Next, create a file named index.html containing the error message you wish to display to your users when the main application is down for maintenance or otherwise not available. This page should reference only media that is guaranteed to be available, no matter the state of your application. For example, if you reference a CSS file that is served by your application, your error page will not work correctly, and your users will see an unstyled error page.

With those steps complete, we can move on to the ELB-related steps. The various methods of setting up an ELB are described in earlier chapters and will not be duplicated here. Refer to [Chapter 6](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#aws_sysadmin_autoscaling) for more information on setting up an Elastic Load Balancer.

For the sake of this example, we will assume you have created an ELB named my-elb, which has a single EC2 instance behind it running your custom application.

The next step is create the failover resource record set that includes a health check. This will periodically check the status of your ELB and application.

First, we must create the primary record set that will be used when the ELB is working as expected. Open the Route 53 Management Console, navigate to your Hosted Zone, click Go To Record Sets, and then click Create Record Set. The resulting screen is shown in [Figure 10-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch10.html#create_primary_failover_recordset).



###### *Figure 10-1. Create primary failover record set*

Some of the values you enter in this screen—such as the Name—will differ from the example screenshot. It is important to choose a low TTL so that failover can occur quickly. If this value is set too high, clients will cache the incorrect value for a longer period of time, resulting in more requests hitting your failed ELB or application.

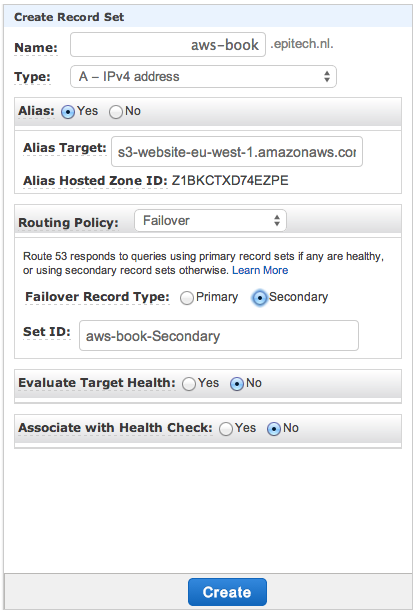
Select the Yes radio button next to Alias, which will cause a new input box to appear. Type the name of your ELB in this new Alias Target input box.

In the Routing Policy drop-down, select Failover and select Primary as the Failover Record Type. Enter a Set ID, such as Primary, to help you remember which record set is which.

Finally, select the Yes radio button next to Evaluate Target Health. Because this record set is an alias to an ELB, Route 53 already has the information it needs to perform the health check against the ELB.

Click Create Record Set to save this information and close the window.

Next, we need to repeat these steps for the secondary resource record set that will be used when the ELB fails its health check. Repeat the previous steps for the secondary resource record set, as shown in [Figure 10-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch10.html#create_secondary_failover_recordset).



###### *Figure 10-2. Create secondary resource record set*

This record set should be configured like the primary record set, with a few changes. The name and TTL values should be the same as in the primary. This record set should also be an alias, but instead of pointing to an ELB, type the name of your S3 bucket in the Alias Target input. Select Failover as the Routing Policy and Secondary as the Failover Record Type. The Set ID should be Secondary, or some other easy-to-remember name.

Click Create Record Set to save the secondary record set.

With those steps completed, the scenario is ready for testing. Visit the domain name you entered as the name for the record sets, such as www.example.com, in your web browser. You should see a page served by your EC2 instance running behind the ELB. If not, go back and recheck the steps to confirm everything is set up as described in the instructions.

If you see your application’s web page, you are ready to test the failover scenario. Terminate the EC2 instance or stop the application process so that the ELB’s health check fails. It can take a few minutes for the failure to be recognized, depending on how you configured the ELB. Remember, you can check the status of the health check in CloudWatch.

Once the health check has failed, visit the web page again. You should now see the error page you uploaded to your S3 bucket. Again, if this does not work, retrace your steps to ensure that everything is configured as described in this section.

Finally, we need to make sure the failover occurs in reverse when your application becomes healthy once more. If you terminated the instance behind the ELB, launch a new one and place this new instance behind the ELB. If you stopped the application process, restart it.

After a few minutes, the health check should recognize that the application has returned to a healthy status. When you refresh the web page, you should once again see a page served by your application.

One potential downside of this approach is that DNS records might be cached by your user’s DNS client or by a caching DNS server that exists on the path between your users and Route 53. Some DNS caching servers do not honor the TTL you chose when creating your record set. Unfortunately, there is no way to work around these misconfigured DNS servers. This can result in the failover appearing to take longer to succeed than it really does, which means these users might see your error page for longer than they should.

Regardless of this, DNS failover provides a useful way of automatically displaying an error or maintenance page when circumstances beyond your immediate control take down your web application. Furthermore, the complete automation of the process means you do not need to worry about putting your error page in place when your application is experiencing problems—instead, you can get on with the more useful task of diagnosing and fixing the problem.

# Recap

* Route 53 is not just a web interface for a BIND-like service. It is a configurable and programmable service, just like the other components of AWS.
* Use Route 53 DNS names as the public face of your application and route traffic to your internal services.
* Be careful when updating DNS records for high-traffic services. A sudden massive increase in traffic could overload your servers. Instead, gradually ramp up traffic by using weight resource record sets.
* Remember to keep your TTLs low when using Route 53 for high availability. Higher TTL values will result in clients using old cached DNS records.