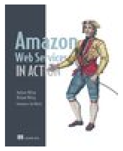


Chapter 12. Decoupling your infrastructure: ELB and SQS

This chapter covers

- The reasons for decoupling a system
- Synchronous decoupling with load balancers
- Asynchronous decoupling with message queues

Imagine that you want some advice on using AWS from us, and therefore we plan to meet in a café. To make this meeting successful, we must

- Be available at the same time
- Be at the same place
- Find each other at the café

The problem with our meeting is that it’s tightly coupled to a location. We can solve that issue by decoupling our meeting from the location, so we change plans and schedule a Google Hangout session. Now we must

- Be available at the same time
- Find each other in Google Hangout

Google Hangout (this also works with all other video/voice chats) does synchronous decoupling. It removes the need to be at the same place while still requiring us to meet at the same time.

We can even decouple from time by using an e-mail conversation. Now we must

- Find each other via email

Email does asynchronous decoupling. You can send an email when the recipient is asleep, and they’ll respond when they’re awake.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don’t run the examples longer than a few days, you won’t pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there are no other things going on in your AWS account. Try to complete the chapter within a few days, because you’ll clean up your account at the end of the chapter.

Note

To fully understand this chapter, you’ll need to have read and understood the concept of auto-scaling covered in chapter 11.

A meeting isn’t the only thing that can be decoupled. In software systems, you can find a lot of tightly coupled components:

- A public IP address is like the location of our meeting. To make a request to

a web server, you must know its public IP address, and the server must be connected to that address. If you want to change the public IP address, both parties are involved in making the appropriate changes.
- If you want to make a request to a web server, the web server must be online at the same time. Otherwise your request will fail. There are many reasons a web server can be offline: someone is installing updates, a hardware failure, and so on.

AWS offers a solution for both of these problems. The *Elastic Load Balancing (ELB)* service provides a load balancer that sits between your web servers and the public internet to decouple your servers synchronously. For asynchronous decoupling, AWS offers a *Simple Queue Service (SQS)* that provides a message queue infrastructure. You'll learn about both services in this chapter. Let's start with ELB.

12.1. Synchronous decoupling with load balancers

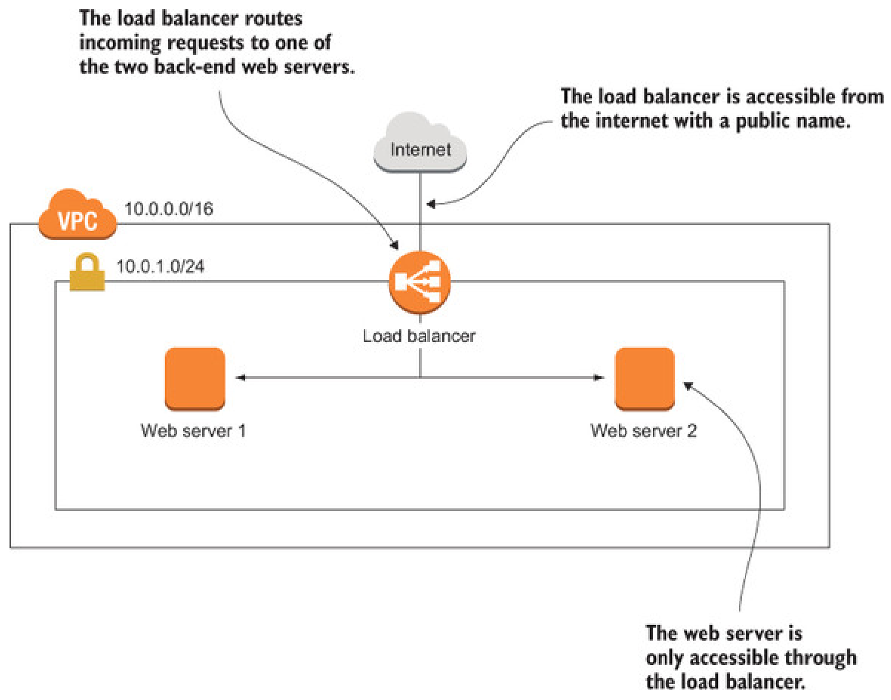
Exposing a single web server to the outside world introduces a dependency: the public IP address of the EC2 instance. From this point on, you can't change the public IP address again because it's used by many clients sending requests to your server. You're faced with the following issues:

- Changing the public IP address is no longer possible because many clients rely on it.
- If you add an additional server (and IP address) to handle increasing load, it's ignored by all current clients: they're still sending all requests to the public IP address of the first server.

You can solve these issues with a DNS name pointing to your server. But DNS isn't fully under your control. DNS servers cache entries, and sometimes they don't respect your time to live (TTL) settings. A better solution is to use a load balancer.

A load balancer can help to decouple a system where the requester awaits an immediate response. Instead of exposing your web servers to the outside world, you only expose the load balancer to the outside world. The load balancer then redirects requests to the web servers behind it. Figure 12.1 shows how this works.

Figure 12.1. A load balancer synchronously decouples your server.


The load balancer routes incoming requests to one of the two back-end web servers.
The load balancer is accessible from the internet with a public name.
Internet
VPC
10.0.0.0/16
10.0.1.0/24
Load balancer
Web server 1
Web server 2
The web server is only accessible through the load balancer.

AWS offers load balancers through the ELB service. The AWS load balancer is fault-tolerant and scalable. For each ELB, you pay $ 0.025 per hour and $ 0.008 per GB of processed traffic. The prices are valid for the North Virginia (us-east-1) region.

Note

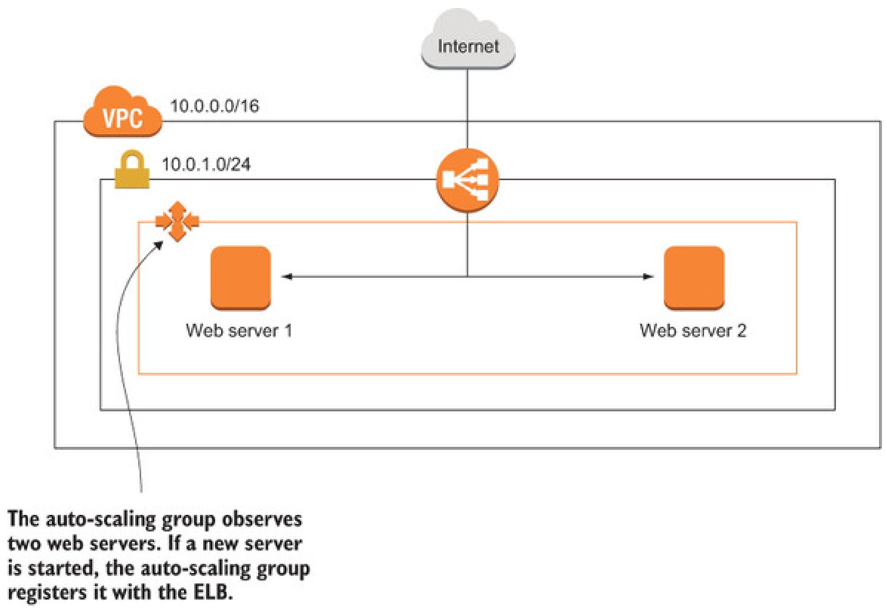
The ELB service doesn't have an independent Management Console. It's integrated into the EC2 service.

A load balancer can be used with more than web servers—you can use load balancers in front of any systems that deal with request/response kind of communication.

12.1.1. Setting up a load balancer with virtual servers

AWS shines when it comes to integrating services together. In chapter 11, you learned about auto-scaling groups. You'll now put a Elastic Load Balancer (ELB) in front of an auto-scaling group to decouple traffic to web servers. The auto-scaling group will make sure you always have two servers running. Servers that are started in the auto-scaling group will automatically register with the ELB. Figure 12.2 shows how the setup will look. The interesting part is that the web servers are no longer accessible directly from the public internet. Only the load balancer is accessible and redirects request to the back-end servers behind it; this is done with security groups, which you learned about in chapter 6.

Figure 12.2. Auto-scaling groups work closely with ELB: they register a new server with the load balancer.


Internet
VPC
10.0.0.0/16
10.0.1.0/24
Web server 1
Web server 2
The auto-scaling group observes two web servers. If a new server is started, the auto-scaling group registers it with the ELB.

An ELB is described by the following:

- The subnets it's attached to. There can be more than one.
- A mapping of the load balancer port to the port on the servers behind the ELB.
- The security groups that are assigned to the ELB. You can restrict traffic to the ELB in the same ways you can with EC2 instances.
- Whether the load balancer should be accessible from the public internet.

The connection between the ELB and the auto-scaling group is made in the auto-scaling group description by specifying `LoadBalancerNames`.

The next listing shows a CloudFormation template snippet to create an ELB and connect it with an auto-scaling group. The listing implements the example shown in figure 12.2.

Listing 12.1. Creating a load balancer and connecting it with an auto-scaling group

```
[...]
"LoadBalancerSecurityGroup": {
  "Type": "AWS::EC2::SecurityGroup",
  "Properties": {
    "GroupDescription": "elb-sg",
    "VpcId": {"Ref": "VPC"},
    "SecurityGroupIngress": [{
      "CidrIp": "0.0.0.0/0",
      "FromPort": 80,
      "ToPort": 80,
      "IpProtocol": "tcp"
    }]
  }
},

"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "Subnet"}],
    "LoadBalancerName": "elb",
    "Listeners": [{
      "InstancePort": "80",
      "InstanceProtocol": "HTTP",
      "LoadBalancerPort": "80",
      "Protocol": "HTTP"
    }],
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internet-facing"
  }
},
"LaunchConfiguration": {
  "Type": "AWS::AutoScaling::LaunchConfiguration",
  "Properties": {
    [...]
  }
},
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",
  "Properties": {
    "LoadBalancerNames": [{"Ref": "LoadBalancer"}],
    "LaunchConfigurationName": {"Ref": "LaunchConfiguration"},
    "MinSize": "2",
    "MaxSize": "2",
    "DesiredCapacity": "2",
    "VPCZoneIdentifier": [{"Ref": "Subnet"}]
  }
}
```

The load balancer only accepts traffic on port 80.

Attaches the ELB to the subnet

Maps the load-balancer port to a port on the servers behind it.

Assigns a security group.

The ELB is publicly accessible (use internal instead of internet-facing to define a load balancer reachable from private network only).

Connects the auto-scaling group with the ELB.

Belongs to MinSize, MaxSize and DesiredCapacity.

To help you explore ELBs, we created a CloudFormation template, located at https://s3.amazonaws.com/awsinaction/chapter12/loadbalancer.json. Create a stack based on that template, and then visit the URL output of your stack with your browser. Every time you reload the page, you should see one of the private IP addresses of a back-end web server.

Cleaning up

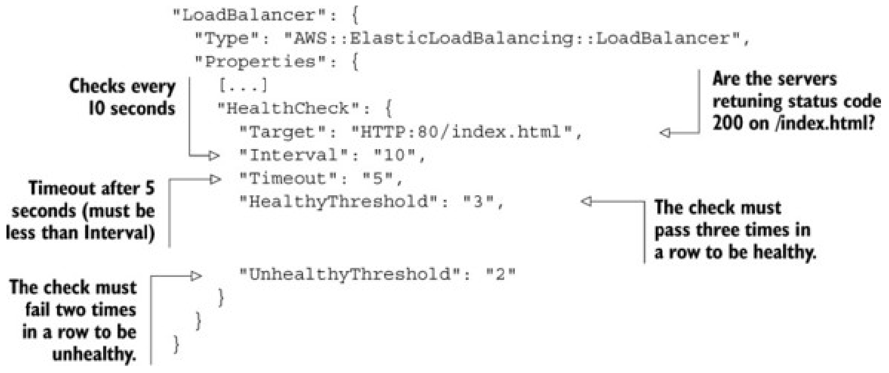
Delete the stack you created.

12.1.2. Pitfall: connecting a server too early

The auto-scaling group is responsible for connecting a newly launched EC2 instance with the load balancer. But how does the auto-scaling group knows when the EC2 instance is installed and ready to accept traffic? Unfortunately, the auto-scaling group doesn't know whether the server is ready; it will register the EC2 instance with the load balancer as soon as the instance is launched. If traffic is sent to a server that's launched but not installed, the request will fail and your users will be unhappy.

But the ELB can send periodic health checks to each server that's connected to find out whether the server can serve requests. In the web server example, you want to check whether you get a status code 200 response for a particular resource, such as /index.html. The following listing shows how this can be done with CloudFormation.

Listing 12.2. ELB health checks to determine whether a server can answer requests



```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
   [...]
   "HealthCheck": {
     "Target": "HTTP:80/index.html",
     "Interval": "10",
     "Timeout": "5",
     "HealthyThreshold": "3",

     "UnhealthyThreshold": "2"
   }
  }
}
```

Checks every 10 seconds
Are the servers retuning status code 200 on /index.html?
Timeout after 5 seconds (must be less than Interval)
The check must pass three times in a row to be healthy.
The check must fail two times in a row to be unhealthy.

Instead of /index.html, you can also request a dynamic page like /healthy.php that does some additional checks to decide whether the web server is ready to handle requests. The contract is that you must return an HTTP status code 200 when the server is ready. That's it.

Aggressive health checks can cause downtime

If a server is too busy to answer a health check, the ELB will stop sending traffic to that server. If the situation is caused by a general load increase on your system, the ELB's response will make the situation worse! We've seen applications experience downtime due to overly aggressive health checks. You need proper load testing to understand what's going on. An appropriate solution is application-specific and can't be generalized.

By default, an auto-scaling group determines if an EC2 instance is healthy based on the heath check that EC2 performs every minute. You can configure an auto-scaling group to use the health check of the load balancer instead. The auto-scaling group will now terminate servers not only if the hardware fails, but also if the application fails. Set `"HealthCheckType": "ELB"` in the auto-scaling group description. Sometimes this setting makes sense because restarting can solve issues like memory, thread pool, or disk overflow, but it can also cause unwanted restarts of EC2 instances in the case of a broken application.

12.1.3. More use cases

So far, you've seen the most common use case for ELB: load-balancing incoming web requests to some web servers over HTTP. As mentioned earlier, ELB can do far more than that. In this section, we'll look at four more typical use cases:

1. ELB can balance TCP traffic. You can place almost any application behind a load balancer.
2. ELB can turn SSL-encrypted traffic into plain traffic if you add your SSL certificate to AWS.
3. ELB can log each request. Logs are stored on S3.
4. ELB can distribute your requests evenly across multiple availability zones.

Handling TCP traffic

Until now, you've only used ELB to handle HTTP traffic. You can also configure ELB to redirect plain TCP traffic, to decouple databases or legacy applications with proprietary interfaces. Compared to an ELB configuration that handles HTTP traffic, you must change `Listeners` and `HealthCheck` to handle TCP traffic with ELB. The health check doesn't check for a specific response as it did when dealing with HTTP; health checks for TCP traffic are healthy when the ELB can open a socket. The following listing shows how you can redirect TCP traffic to MySQL back ends.

Listing 12.3. ELB handling plain TCP traffic (not only HTTP)

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "SubnetA"}, {"Ref": "SubnetB"}],
    "LoadBalancerName": "elb",
    "Listeners": [{
      "InstancePort": "3306",
      "InstanceProtocol": "TCP",
      "LoadBalancerPort": "3306",
      "Protocol": "TCP"
    }],
    "HealthCheck": {
      "Target": "TCP:3306",  )
      "Interval": "10",
      "Timeout": "5",
      "HealthyThreshold": "3",
      "UnhealthyThreshold": "2"
    },
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internal"
  }
}
```

Redirects traffic on port 3306 (MySQL) to the back-end servers

Healthy when ELB can open a socket on port 3306 on the back-end server

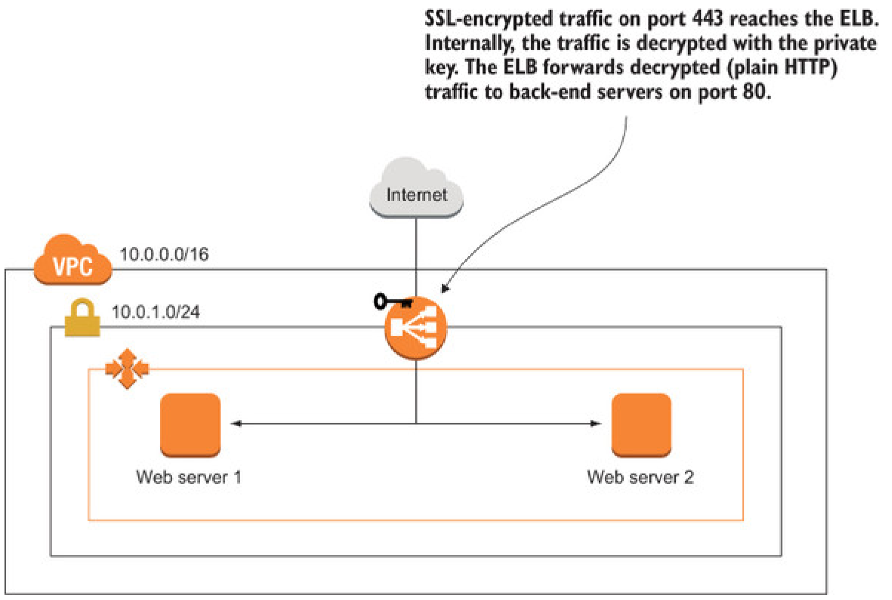
The MySQL database shouldn't be public; choose an internal load balancer.

You can also configure port 80 to be handled as TCP traffic, but you'll lose the ability to do health checks based on the status code that's returned by your web server.

Terminating SSL

An ELB can be used to terminate SSL without the need to do the configuration on your own. Terminating SSL means the ELB offers an SSL-encrypted endpoint that forwards requests unencrypted to your back-end servers. Figure 12.3 shows how this works.

Figure 12.3. A load balancer can accept encrypted traffic, decrypt the traffic, and forward unencrypted traffic to the back end.


SSL-encrypted traffic on port 443 reaches the ELB. Internally, the traffic is decrypted with the private key. The ELB forwards decrypted (plain HTTP) traffic to back-end servers on port 80.
Internet
VPC
10.0.0.0/16
10.0.1.0/24
Web server 1
Web server 2

You can use predefined security policies from AWS to get a secure SSL configuration that takes care of SSL vulnerabilities in the wild. You can accept requests on port 443 (HTTPS); the ELB terminates SSL and forwards the request to port 80 on a web server. That's an easy solution to offer SSL-encrypted communication. SSL termination doesn't just work for HTTP requests; it also works for TCP traffic (such as POP3, SMTP, FTP).

Note

The following example only works if you already own an SSL certificate. If you don't, you need to buy an SSL certificate or skip the example. AWS doesn't offer

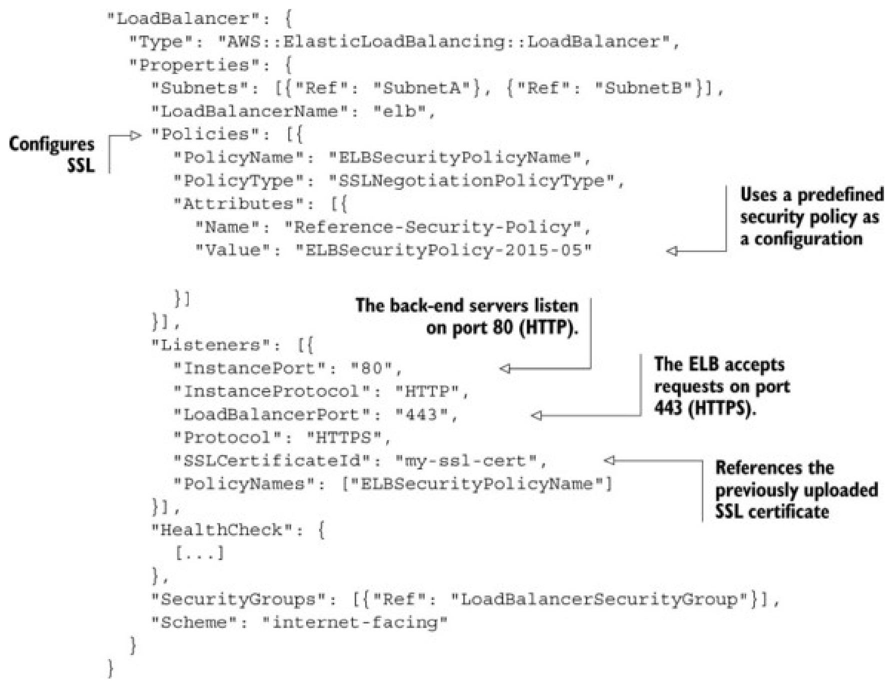
SSL certificates at the moment. You could use a self-signed certificate for testing purposes.

Before you can activate SSL encryption, you must upload your SSL certificate to IAM with the help of the CLI:

```
$ aws iam upload-server-certificate \
--server-certificate-name my-ssl-cert \
--certificate-body file://my-certificate.pem \
--private-key file://my-private-key.pem \
--certificate-chain file://my-certificate-chain.pem
```

Now you can use your SSL certificate by referencing `my-ssl-cert`. The following listing shows how encrypted HTTP communication can be configured with the help of the ELB.

Listing 12.4. Terminating SSL with ELB to offer encrypted communication



```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    "Subnets": [{"Ref": "SubnetA"}, {"Ref": "SubnetB"}],
    "LoadBalancerName": "elb",
    "Policies": [{
      "PolicyName": "ELBSecurityPolicyName",
      "PolicyType": "SSLNegotiationPolicyType",
      "Attributes": [{
        "Name": "Reference-Security-Policy",
        "Value": "ELBSecurityPolicy-2015-05"

      }]
    }],
    "Listeners": [{
      "InstancePort": "80",
      "InstanceProtocol": "HTTP",
      "LoadBalancerPort": "443",
      "Protocol": "HTTPS",
      "SSLCertificateId": "my-ssl-cert",
      "PolicyNames": ["ELBSecurityPolicyName"]
    }],
    "HealthCheck": {
      [...]
    },
    "SecurityGroups": [{"Ref": "LoadBalancerSecurityGroup"}],
    "Scheme": "internet-facing"
  }
}
```

Configures SSL
Uses a predefined security policy as a configuration
The back-end servers listen on port 80 (HTTP).
The ELB accepts requests on port 443 (HTTPS).
References the previously uploaded SSL certificate

Terminating SSL with the help of the ELB eliminates many administrative tasks that are critical to providing secure communication. We encourage you to offer HTTPS with the help of an ELB to protect your customers from all kinds of attacks while they're communicating with your servers.

Warning

It's likely that the security policy `ELBSecurityPolicy-2015-05` is no longer the most up-to-date. The security policy defines what versions of SSL are supported, what ciphers are supported, and other security-related options. If you aren't using the latest security policy version, your SSL setup is probably vulnerable. Visit http://mng.bz/916U to get the latest version.

We recommend that you offer only SSL-encrypted communication to your users. In addition to protecting sensitive data, it also has a positive impact on Google rankings.

Logging

ELB can integrate with S3 to provide access logs. Access logs contain all the requests processed by the ELB. You may be familiar with access logs from web servers like Apache web server; you can use access logs to debug problems with your back end and analyze how many requests have been made to your system.

To activate access logging, the ELB must know to which S3 bucket logs should be written. You can also specify how often the access logs should be written to

S3. You need to set up an S3 bucket policy to allow the ELB to write to the bucket, as shown in the following listing.

Listing 12.5. policy.json

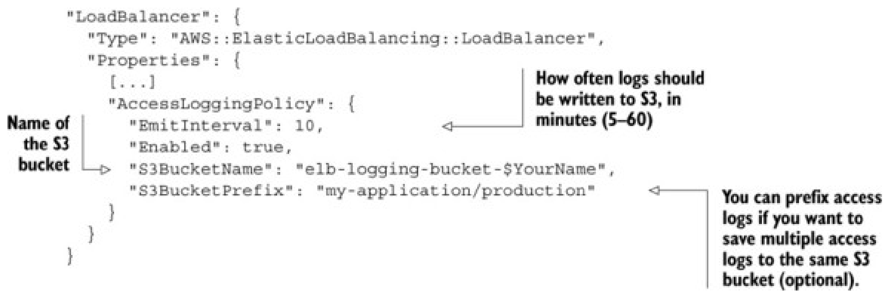
```
{
  "Id": "Policy1429136655940",
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Stmt1429136633762",
    "Action": ["s3:PutObject"],
    "Effect": "Allow",
    "Resource": "arn:aws:s3:::elb-logging-bucket-$YourName/*",
    "Principal": {
      "AWS": [
        "127311923021", "027434742980", "797873946194",
        "156460612806", "054676820928", "582318560864",
        "114774131450", "783225319266", "507241528517"
      ]
    }
  }]
}
```

To create the S3 bucket with the policy, use the CLI—but don't forget to replace `$YourName` with your name or nickname to prevent name clashes with other readers. This also applies to the policy.json file. To save some time, you can download the policy from https://s3.amazonaws.com/awsinaction/chapter12/policy.json:

```
$ aws s3 mb s3://elb-logging-bucket-$YourName
$ aws s3api put-bucket-policy --bucket elb-logging-bucket-$YourName \
--policy file://policy.json
```

You can activate access logging with the following CloudFormation description.

Listing 12.6. Activating access logs written by ELB



```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "AccessLoggingPolicy": {
      "EmitInterval": 10,
      "Enabled": true,
      "S3BucketName": "elb-logging-bucket-$YourName",
      "S3BucketPrefix": "my-application/production"
    }
  }
}
```

How often logs should be written to S3, in minutes (5–60)
Name of the S3 bucket
You can prefix access logs if you want to save multiple access logs to the same S3 bucket (optional).

The ELB will now write access-log files to the specified S3 bucket from time to time. The access log is similar to the one created by the Apache web server, but you can't change the format of the information it contains. The following snippet shows a single line of an access log:

```
2015-06-23T06:40:08.771608Z elb 92.42.224.116:17006 172.31.38.190:80
0.000063 0.000815 0.000024 200 200 0 90
"GET http://elb-....us-east-1.elb.amazonaws.com:80/ HTTP/1.1"
"Mozilla/5.0 (Macintosh; ...) Gecko/20100101 Firefox/38.0" - -
```

Here are examples of the pieces of information an access log always contains:

- Time stamp: `2015-06-23T06:40:08.771608Z`
- Name of the ELB: `elb`
- Client IP address and port: `92.42.224.116:17006`
- Back-end IP address and port: `172.31.38.190:80`
- Number of seconds the request was processed in the load balancer: `0.000063`
- Number of seconds the request was processed in the back end: `0.000815`
- Number of seconds the response was processed in the load balancer: `0.000024`
- HTTP status code returned by the load balancer: `200`
- HTTP status code returned by back end: `200`
- Number of bytes received: `0`
- Number of bytes sent: `90`
- Request: `"GET http://elb-....us-east-1.elb.amazonaws.com:80/ HTTP/1.1"`
- User agent: `"Mozilla/5.0 (Macintosh; ...) Gecko/20100101 Firefox/38.0"`

Cleaning up

Remove the S3 bucket you created in the logging example:

```
$ aws s3 rb --force s3://elb-logging-bucket-$YourName
```

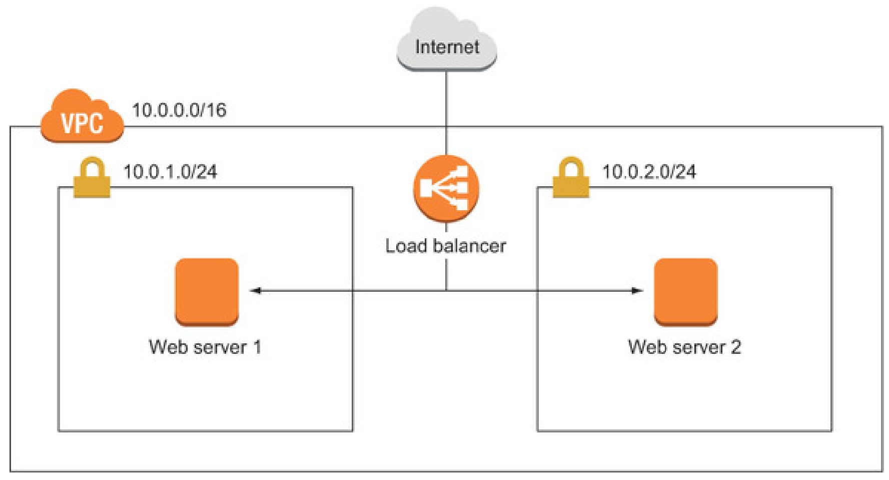
Cross-zone load balancing

The ELB is a fault-tolerant service. If you create an ELB, you receive a public name like `elb-1079556024.us-east-1.elb.amazonaws.com` as the endpoint. It's interesting to see what's behind that name. You can use the command-line application `dig` (or `nslookup` on Windows) to ask a DNS server about a particular name:

```
$ dig elb-1079556024.us-east-1.elb.amazonaws.com
[...]
;; ANSWER SECTION:
elb-1079556024.us-east-1.elb.amazonaws.com. 42 IN A 52.0.40.9
elb-1079556024.us-east-1.elb.amazonaws.com. 42 IN A 52.1.152.202
[...]
```

The name `elb-1079556024.us-east-1.elb.amazonaws.com` resolves to two IP addresses: 52.0.40.9 and 52.1.152.202. When you create a load balancer, AWS starts two instances in the background and uses DNS to distributed between the two. To make the servers fault-tolerant, AWS spawns the load-balancer instances in different availability zones. By default, each load-balancer instance of the ELB sends traffic only to EC2 instances in the same availability zone. If you want to distribute requests across availability zones, you can enable *cross-zone load balancing*. Figure 12.4 shows a scenario in which cross-zone load balancing is important.

Figure 12.4. Enabling cross-zone load balancing to distribute traffic between availability zones



The following CloudFormation snippet shows how this can be activated:

```
"LoadBalancer": {
  "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties": {
    [...]
    "CrossZone": true
  }
}
```

We recommend that you enable cross-zone load balancing, which is disabled by default, to ensure that requests are routed evenly across all back-end servers.

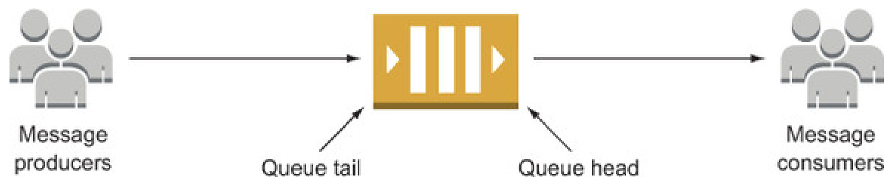
In the next section, you'll learn more about asynchronous decoupling.

12.2. Asynchronous decoupling with message queues

Synchronous decoupling with ELB is easy; you don't need to change your code to do it. But for asynchronous decoupling, you have to adapt your code to work with a message queue.

A message queue has a head and a tail. You can add new messages to the tail while reading messages from the head. This allows you to decouple the production and consumption of messages. The producers and consumers don't know each other; they both only know about the message queue. Figure 12.5 illustrates this principle.

Figure 12.5. Producers send messages to a message queue, and consumers read messages.



You can put new messages onto the queue while no one is reading messages, and the message queue acts as a buffer. To prevent message queues from growing infinitely large, messages are only saved for a certain amount of time. If you consume a message from a message queue, you must acknowledge the successful processing of the message to permanently delete it from the queue.

The Simple Queue Service (SQS) is a fully managed AWS service. SQS offers message queues that guarantee the delivery of messages at least once:

- Under rare circumstances, a single message will be available for consumption twice. This may sound strange if you compare it to other message queues, but you'll see how to deal with this problem later in the chapter.
- SQS doesn't guarantee the order of messages, so you may read messages in a different order than they were produced.

This limitation of SQS is also beneficial:

- You can put as many messages into SQS as you like.
- The message queue scales with the number of messages you produce and consume.

The pricing model is also simple: you pay $0.00000050 per request to SQS or $0.5 per million requests. Producing a message is one request, and consuming is another request (if your payload is larger than 64 KB, every 64 KB chunk counts as one request).

12.2.1. Turning a synchronous process into an asynchronous one

A typical synchronous process looks like this: a user makes a request to your server, something happens on the server, and a result is returned to the user. To make things more concrete, we'll talk about the process of creating a preview image of an URL in the following example:

1. The user submits a URL.
2. The server downloads the content at the URL and converts it into a PNG image.
3. The server returns the PNG to the user.

With one small trick, this process can be made asynchronous:

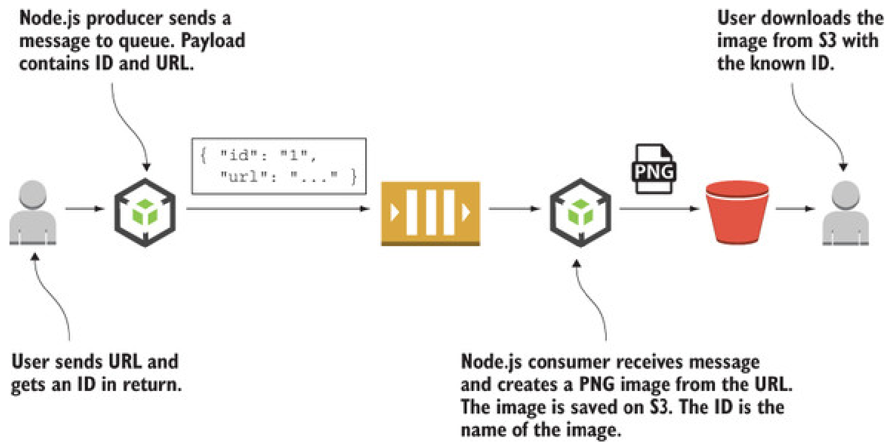
1. The user submits a URL.
2. The server puts a message onto a queue that contains a random ID and the URL.
3. The server returns a link to the user where the PNG image will be found in the future. The link contains the random ID (http://$Bucket.s3-website-us-east-1.amazonaws.com/$RandomId.png).
4. In the background, a worker consumes the message from the queue, downloads the content, converts the content into a PNG, and uploads the image to S3.
5. At some point in time, the user tries to download the PNG at the known location.

If you want to make a process asynchronous, you must manage the way the process initiator tracks the status of the process. One way of doing that is to return an ID to the initiator that can be used to look up the process. During the process, the ID is passed from step to step.

12.2.2. Architecture of the URL2PNG application

You'll now create a simple but decoupled piece of software named URL2PNG that converts the URL of a web page into a PNG. Again, you'll use Node.js to do the programming part, and you'll use SQS. Figure 12.6 shows how the URL2PNG application works.

Figure 12.6. How the URL2PNG application works


Node.js producer sends a message to queue. Payload contains ID and URL.
User downloads the image from S3 with the known ID.
{ "id": "1", "url": "..." }
PNG
User sends URL and gets an ID in return.
Node.js consumer receives message and creates a PNG image from the URL. The image is saved on S3. The ID is the name of the image.

To complete the example, you need to create an S3 bucket with web hosting enabled. Execute the following commands, replacing `$YourName` with your name or nickname to prevent name clashes with other readers:

```
$ aws s3 mb s3://url2png-$YourName
$ aws s3 website s3://url2png-$YourName --index-document index.html \
--error-document error.html
```

Web hosting is needed so users can later download the images from S3. Now it's time to create the message queue.

12.2.3. Setting up a message queue

Creating an SQS queue is simple—you only need to specify the name of the queue:

```
$ aws sqs create-queue --queue-name url2png
{
  "QueueUrl": "https://queue.amazonaws.com/878533158213/url2png"
}
```

The returned `QueueUrl` is needed later in the example, so be sure to save it.

12.2.4. Producing messages programmatically

You now have an SQS queue to send messages to. To produce a message, you need to specify the queue and a payload. You'll again use Node.js in combination with the AWS SDK to connect your program with AWS.

Installing and getting started with Node.js

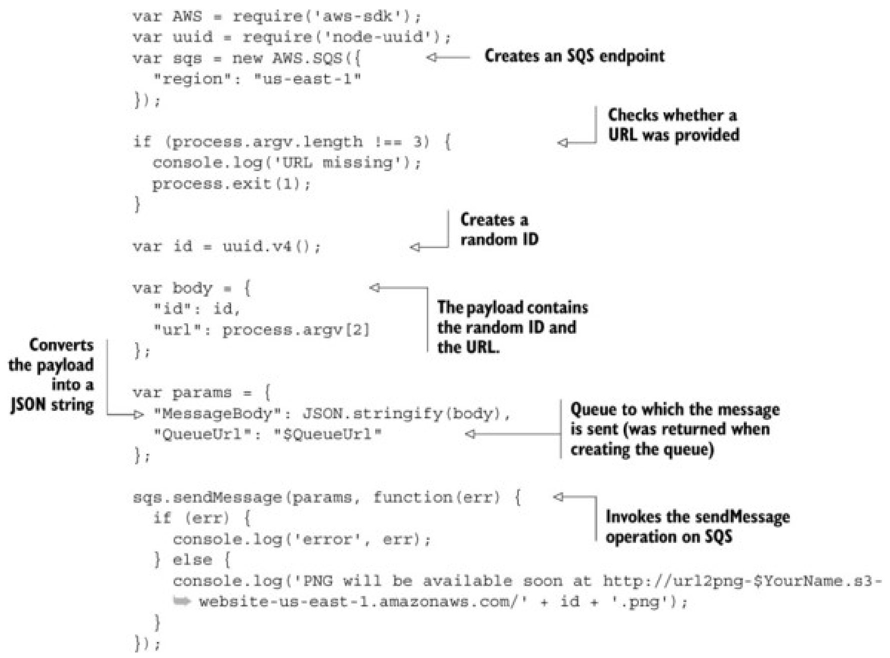
To install Node.js, visit https://nodejs.org and download the package that fits your OS.

Here's how the message is produced with the help of the AWS SDK for Node.js; it will later be consumed by the URL2PNG worker. The Node.js script can then be used like this (don't try to run this command now—you need to install and configure URL2PNG first):

```
$ node index.js "http://aws.amazon.com"
PNG will be available soon at
http://url2png-$YourName.s3-website-us-east-1.amazonaws.com/XYZ.png
```

As usual, you'll find the code in the book's code repository on GitHub https://github.com/AWSinAction/code. The URL2PNG example is located at /chapter12/url2png/. The following listing shows the implementation of index.js.

Listing 12.7. index.js: sending a message to the queue



```
var AWS = require('aws-sdk');
var uuid = require('node-uuid');
var sqs = new AWS.SQS({
  "region": "us-east-1"
});

if (process.argv.length !== 3) {
  console.log('URL missing');
  process.exit(1);
}

var id = uuid.v4();

var body = {
  "id": id,
  "url": process.argv[2]
};

var params = {
  "MessageBody": JSON.stringify(body),
  "QueueUrl": "$QueueUrl"
};

sqs.sendMessage(params, function(err) {
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be available soon at http://url2png-$YourName.s3-
    ➥ website-us-east-1.amazonaws.com/' + id + '.png');
  }
});
```

Creates an SQS endpoint
Checks whether a URL was provided
Creates a random ID
The payload contains the random ID and the URL.
Converts the payload into a JSON string
Queue to which the message is sent (was returned when creating the queue)
Invokes the sendMessage operation on SQS

Before you can run the script, you need to install the Node.js modules. Run `npm install` in your terminal to install the dependencies. You'll find a config.json file that needs to be modified. Make sure to change `QueueUrl` to the queue you created at the beginning of this example and change `Bucket` to `url2png-$YourName`.

Now you can run the script with `node index.js "http://aws.amazon.com"`. The program should response with something like "PNG will be available soon at http://url2png-$YourName.s3-website-us-east-1.amazonaws.com/XYZ.png". To verify that the message is ready for consumption, you can ask the queue how many messages are inside:

```
$ aws sqs get-queue-attributes \
--queue-url $QueueUrl \
--attribute-names ApproximateNumberOfMessages
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}
```

Next, it's time to work on the worker that consumes the message and does all the work of generating a PNG.

12.2.5. Consuming messages programmatically

Processing a message with SQS takes three steps:

1. Receive a message.
2. Process the message.
3. Acknowledge successful message processing.

You'll now implement each of these steps to change a URL into a PNG.

To receive a message from an SQS queue, you must specify the following:

- The queue
- The maximum number of messages you want to receive. To get higher throughput, you can get batches of messages.
- The number of seconds you want to take this message from the queue to process it. Within that time, you must delete the message from the queue, or it will be received again.
- The maximum number of seconds you want to wait to receive messages. Receiving messages from SQS is done by polling the API. But the API allows long-polling for a maximum of 10 seconds.

The next listing shows how this is done with the SDK.

Listing 12.8. worker.js: receiving a message from the queue

```
var fs = require('fs');
var AWS = require('aws-sdk');
var webshot = require('webshot');
var sqs = new AWS.SQS({
  "region": "us-east-1"
});
var s3 = new AWS.S3({
  "region": "us-east-1"
});

function receive(cb) {
  var params = {
    "QueueUrl": "$QueueUrl",
    "MaxNumberOfMessages": 1,
    "VisibilityTimeout": 120,
    "WaitTimeSeconds": 10

  };
  sqs.receiveMessage(params, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Messages === undefined) {
        cb(null, null);
      } else {
        cb(null, data.Messages[0]);
      }
    }
  });
}
```

Takes the message from the queue for 120 seconds

Consumes not more than one message at once

Long poll for 10 seconds to wait for new messages

Invokes the receiveMessage operation on SQS

Checks whether a message is available

Gets the one and only message

The receive step has now been implemented. The next step is to process the message. Thanks to a Node.js module called `webshot`, it's easy to create a screenshot of a website.

Listing 12.9. worker.js: processing a message (take screenshot and upload to S3)

```
function process(message, cb) {
  var body = JSON.parse(message.Body);
  var file = body.id + '.png';
  webshot(body.url, file, function(err) {
    if (err) {
      cb(err);
    } else {
      fs.readFile(file, function(err, buf) {
        if (err) {
          cb(err);
        } else {
          var params = {
            "Bucket": "url2png-$YourName",
            "Key": file,
            "ACL": "public-read",
            "ContentType": "image/png",

            "Body": buf
          };
          s3.putObject(params, function(err) {
            if (err) {
              cb(err);
            } else {
              fs.unlink(file, cb);
            }
          });
        }
      });
    }
  });
}
```

Creates the screenshot with the webshot module

The message body is a JSON string. You convert it back into a JavaScript object.

Opens the screenshot that was saved to local disk by the webshot module

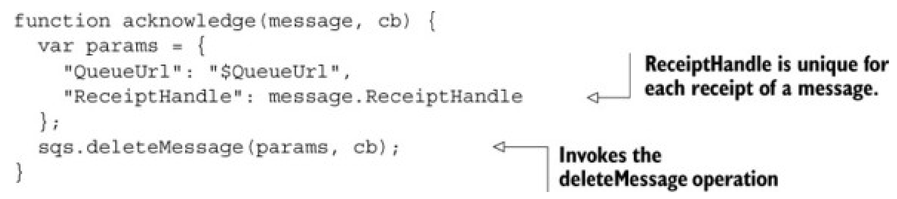
Allows everyone to read the screenshot on S3

Uploads the screenshot to S3

Removes the screenshot from local disk

The only step that's missing is to acknowledge that the message was successfully consumed. If you receive a message from SQS, you get a `ReceiptHandle`, which is a unique ID that you need to specify when you delete a message from a queue.

Listing 12.10. worker.js: acknowledging a message (deletes the message from the queue)

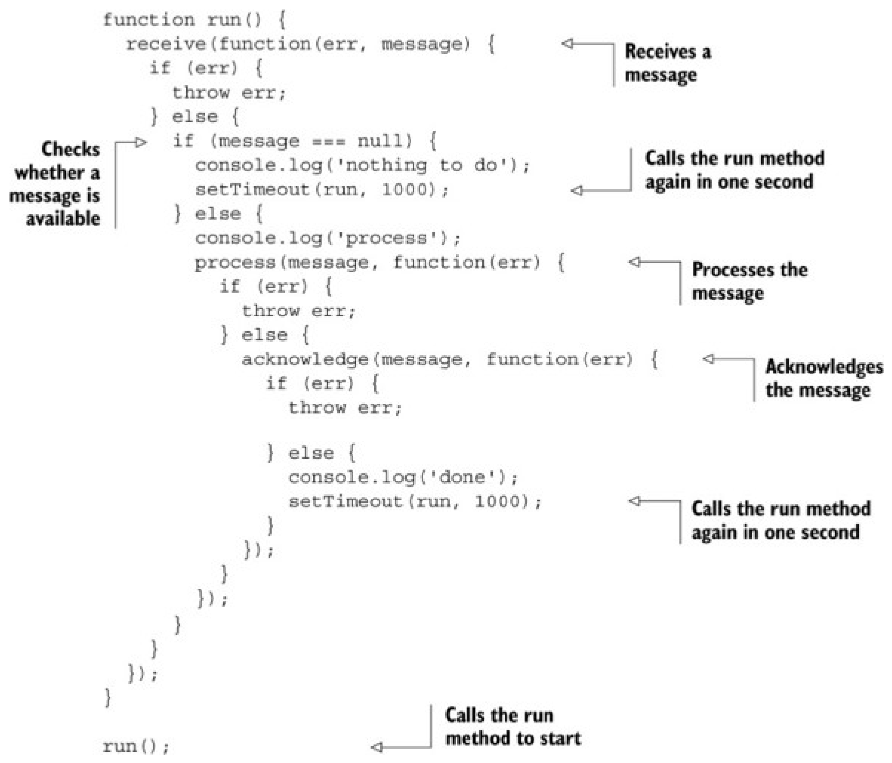


```
function acknowledge(message, cb) {
  var params = {
    "QueueUrl": "$QueueUrl",
    "ReceiptHandle": message.ReceiptHandle
  };
  sqs.deleteMessage(params, cb);
}
```

ReceiptHandle is unique for each receipt of a message.
Invokes the deleteMessage operation

You have all the parts; now it's time to connect them.

Listing 12.11. worker.js: connecting the parts



```
function run() {
  receive(function(err, message) {
    if (err) {
      throw err;
    } else {
      if (message === null) {
        console.log('nothing to do');
        setTimeout(run, 1000);
      } else {
        console.log('process');
        process(message, function(err) {
          if (err) {
            throw err;
          } else {
            acknowledge(message, function(err) {
              if (err) {
                throw err;

              } else {
                console.log('done');
                setTimeout(run, 1000);
              }
            });
          }
        });
      }
    }
  });
}

run();
```

Receives a message
Checks whether a message is available
Calls the run method again in one second
Processes the message
Acknowledges the message
Calls the run method again in one second
Calls the run method to start

Now you can start the worker to process the message that is already in the queue. Run the script with `node worker.js`. You should see some output that says the worker is in the process step and that then switches to Done. After a few seconds, the screenshot should be uploaded to S3. Your first asynchronous application is complete.

You've created an application that is asynchronously decoupled. If the URL2PNG service becomes popular and millions of users start using it, the queue will become longer and longer because your worker can't produce that many PNGs from URLs. The cool thing is that you can add as many workers as you like to consume those messages. Instead of only 1 worker, you can start 10 or 100. The other advantage is that if a worker dies for some reason, the message that was in flight will become available for consumption after two minutes and will be picked up by another worker. That's fault-tolerant! If you design your system asynchronously decoupled, it's easy to scale and a good foundation to be fault-tolerant. The next chapter will concentrate on this topic.

Cleaning up

Delete the message queue as follows:

```
$ aws sqs delete-queue --queue-url $QueueUrl
```

And don't forget to clean up and delete the S3 bucket used in the example. Issue the following command, replacing `$YourName` with your name:

```
$ aws s3 rb --force s3://url2png-$YourName
```

12.2.6. Limitations of messaging with SQS

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail.

SQS doesn't guarantee that a message is delivered only once

If a received message isn't deleted within `VisibilityTimeout`, the message will be received again. This problem can be solved by making the receive idempotent. *Idempotent* means that no matter how often the message is consumed, the result stays the same. In the URL2PNG example, this is true by design: if you process the message multiple times, the same image is uploaded to S3 multiple times. If the image is already available on S3, it's replaced. Idempotence solves many problems in distributed systems that guarantee at least single delivery of messages.

Not everything can be made idempotent. Sending an e-mail is a good example: if you process a message multiple times and it sends an email each time, you'll annoy the addressee. As a workaround, you can use a database to track whether you already sent the email.

In many cases, at least once is a good trade-off. Check your requirements before using SQS if this trade-off fits your needs.

SQS doesn't guarantee the message order

Messages may be consumed in a different order than the order in which you produced them. If you need a strict order, you should search for something else. SQS is a fault-tolerant and scalable message queue. If you need a stable message order, you'll have difficulty finding a solution that scales like SQS. Our advice is to change the design of your system so you no longer need the stable order or produce the order at the client side.

SQS doesn't replace a message broker

SQS isn't a message broker like ActiveMQ—SQS is only a message queue. Don't expect features like those offered by message brokers. Considering SQS versus ActiveMQ is like comparing DynamoDB to MySQL.

12.3. Summary

- Decoupling makes things easier because it reduces dependencies.
- Synchronous decoupling requires two sides to be available at the same time, but the sides don't know each other.
- With asynchronous decoupling, you can communicate without both sides being available.
- Most applications can be synchronously decoupled without touching the code, using the load balancer offered by the Elastic Load Balancing service.
- A load balancer can make periodic health checks to your application to determine whether the back end is ready to serve traffic.
- Asynchronous decoupling is only possible with asynchronous processes. But you can modify a synchronous process to be an asynchronous one most of the time.
- Asynchronous decoupling with SQS requires programming against SQS with one of the SDKs.