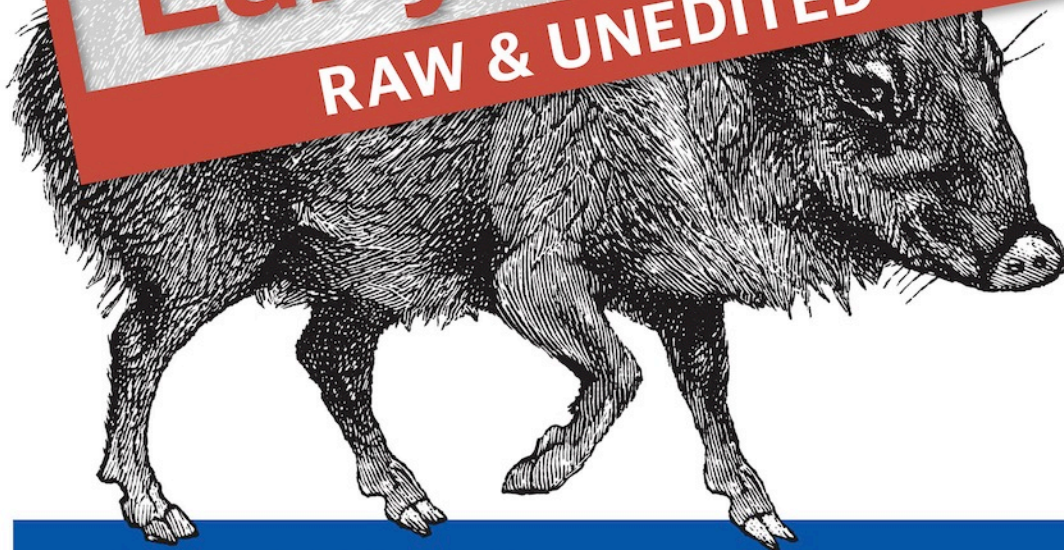O'REILLY®

Early Release

RAW & UNEDITED

AWS System Administration

BEST PRACTICES FOR SYSTEM ADMINISTRATORS
IN THE AMAZON CLOUD

Mike Ryan

AWS System Administration

Mike Ryan

AWS System Administration

by Mike Ryan

Table of Contents

Preface

System administration is a complicated topic that requires practitioners to be familiar with an ever-expanding range of applications and servieces. In some ways Amazon Web Services is just another tool to add to your toolkit, yet it can also be considered a discipline in and of itself. Successfully building and deploying infrastructure on AWS involves a thorough understanding of the underling operating system concerns, software architecture and delivery practices, as well as the myriad components that comprise Amazon Web Services.

I run a DevOps consultancy, helping startups and small businesses reap the benefits of tools and processes that were previously only available to organisations with large teams of systems administrators. Many of these businesses do not have a dedicated systems administrator, and the development team is responsible for deploying and maintaining the architecture.

In working with these clients I noticed a number of patterns in how people were working with AWS. Those that came from a pure development background (i.e. without sysadmin experience) would often build an infrastructure that left out many of the things sysadmins would take for granted, such as monitoring and logging. The lack of monitoring and logging would then make it difficult to track down issues, leading to more downtime than was necessary.

At the other end of the spectrum were those with a lot of sysadmin experience but lesser or no development experience. This group was more likely to treat AWS as nothing more than a virtual machine hosting provider, simply using EC2 to run a fleet of static instances without taking advantage of any high-availability features such as Auto Scaling and Elastic Load Balancers. This is akin to buying a Ferrari and then only using it to buy groceries once per week: Fun, but not cost-effective.

Using AWS is a fundementally different mindset to deploying groups of static servers. You do not simply set up some servers and then periodically perform maintenance. Instead, you use the AWS toolset (automatic instance replacement, scaling up and down

in response to demand, et cetera) to build a system. In this sense it is more like programming than traditional system administration.

The aim of this book is to help people reach a compromise between these two approaches, and help you make the right choice for your application's specific hosting requirements.

If you are a developer this book will give you enough system administration knowledge to ensure you are using AWS effectively, and help you build a robust and resilient application infrastructure.

For systems administrators it will show you how you can keep your favourite tools and processes while working with AWS, and hopefully save you from reinventing some wheels along the way.

AWS is a collection of cloud computing services that can be combined to build scalable and reliable applications and services. It comprises a number of different components, each with their names and configuration options, which are offered under the AWS umbrella. Some of these—such as EC2 and S3—are extremely popular and well-known. Others, such as Kinesis and CloudFormation, are less well known.

Because covering each of these services in detail would result in a multi-volume tome of formiddable size, this book will focus on the more commonly used services and provide a jumping-off point for learning about the others.

If you are familiar with AWS, feel free to hop between chapter to find the information that is most interesting or relevant to your current project. Beginners to AWS should work through the book sequentially, as each chapter builds on information presented in the previous chapters.

Chapter 2 introduces what is perhaps the most well known of all AWS services, EC2. This chapter also introduces my personal favorite AWS service, CloudFormation. In Chapter 3 we look at some of the security features offered by AWS.

Chapter 4 introduces configuration management tools, a common requirement when automating a cloud infrastructure. Using these tools, Chapter and Chapter 6 demonstrate the process of deploying an example application to AWS.

Chapter 7 looks at some of the different methods of deploying application and infrastructure updates to your environment. Chapter 8 builds on this and discusses the creation of reusable components to save time.

Log management, a more traditional sysadmin task that has some interesting implications in the cloud, is the topic of Chapter 9.

Chapter 10 covers another traditional sysadmin task: DNS with Amazon's Route53 service.

Monitoring with Amazon's CloudWatch service and other monitoring tools is discussed in Chapter 11.

Finally, Chapter 12 looks as some of the ways of backing up your data both in and outside of the Amazon cloud.

Audience

This book is written for system administrators and developers. I assume you are comfortable with the basic tools used to administer the operating system and common services such as DNS. If you plan to use Puppet or Chef for automation, you need to learn basic information about those tools elsewhere. A working knowledge of Git or other source code management system. I do not expect you to have prior knowledge of AWS or other virtualization and cloud products.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, and environment variables.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> This icon signifies a tip, suggestion, or general note.

> This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. Major examples can be downloaded from my GitHub repository. Many other small examples are scattered through the book; I have not bothered to include them in the repository because they are fairly easy to type in. Further examples and resources can be found at www.awssystemadministration.com.

In general, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*AWS System Administration* by Mike Ryan (O'Reilly). Copyright 2014 Mike Ryan, 9781449342579."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> 800-998-9938 (in the United States or Canada)
> 707-829-0515 (international or local)
> 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/9781449342579*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgements

This book would not exist without the help and support of a lot of amazing people. I would like to thank my family for their love and support which turned me in to the geek I am today. I would also like to thank Cynthia Stolk for providing me with endless cups of coffee to fuel this book, and Rachel Kersten and Rebecca Lay for making sure it actually got finished.

Thanks are due to friends and colleagues who gave feedback and suggestions: Bartek Swedrowski, Dave Letorey, Guyon Morée, Jurg van Vliet, Keith Perhac, Peter van Kampen, Rick van Hattem, Ross Gynn, Sofie Pelmelay and Thierry Schellenbach. Either directly or indirectly, you helped shape this book.

Finally, I would like to thank the excellent team at O'Reilly for making this happen. Particular thanks are due to my editor Andy Oram for persevering on the long road. And of course, this book would be empty if not for the team behind Amazon Web Services.

Setting Up AWS Tools

The role of the system administrator is changing. Just a few years ago, most sysadmins dealt with server farms of physical hardware and performing detailed capacity planning. Scaling up your application meant ordering new hardware and perhaps spending time racking it up in the datacenter. Now there is a huge section of the industry that has never touched physical hardware. We scale up by issuing an API call or clicking a button in a web page to bring new capacity online.

Although the term has been co-opted by marketers, the cloud is an amazing thing. In this context I am using "cloud" to refer to the idea of scalable, on-demand computing and application services, rather than "cloud-based" services like Google Mail.

As more competition enters the cloud market space, its appeal for sysadmins and business owners alike is increasing on an almost daily basis. Amazon Web Services continues to drive the cloud computing market forward by frequently introducing new tools and services (in fact they are introduced with such regularity that writing a book about them is almost the literary equivalent of Achilles and the tortoise).

Economies of scale are constantly pushing down the price of cloud services. Although environments like AWS or Google Compute Cloud are not suitable for every application, it is becoming increasingly clear that cloud skills are becoming a required part of a well-rounded sysadmin's toolkit.

For businesses, the cloud opens up new avenues of flexibility. Tech teams can do things that would have been prohibitively expensive just a few years ago. The games and applications that are lucky enough to become runaway hits often require a high amount of backend computing capacity. Bringing this capacity online in hours rather than weeks enables these companies to quickly respond to success, without requiring multi-year lease commitments or upfront capital expenditure.

In the age of the Lean Startup, developers and managers know how important it is to quickly iterate and improve their application code. Services like AWS allow you to treat

your infrastructure the same way, letting a relatively small team manage massively scalable application infrastructures.

Preparing Your Tools

There are a few different ways to manage your AWS infrastructure components. The Web Console is the first interface most users see. Although great for exploring and learning about the services, it does not lend itself to automation.

The AWS APIs are a collection of API endpoints that can be used to manage AWS services from your own application. There are implementations in many popular programming languages and platforms, which can be downloaded from the AWS site.

awscli is a command line tool released by Amazon. It can be used to control the vast majority of AWS components from the command line, making it suitable to use in automated build systems and scripts. Before `awscli` was released, Amazon provided a separate management tool for each service. That is, EC2 was managed by one program and SQS by another. The tools did not all use a consistent naming convention for parameters, making them less convenient to use.

There are a few actions that cannot, at the time of writing, be performed by the awscli tool. For this reason, you might find it necessary to install the previous versions. Installation instructions follow.

Installing the AWS Command Line Tool

First, the installation process for awscli. Because it is a Python package, it can be installed with *pip*, the Python package management tool. This is included on many systems by default, but you might need to install it manually. On Debian systems this can be done with:

```
sudo apt-get install python-pip
```

Once you have *pip* on your system, the installation is incredibly simple:

```
pip install awscli
```

You might need to first install the Pyth

Once installed, run *aws help* to get an idea of the features this tool provides. For example:

Action	Command
`aws ec2 run-instances`	Launch one or more EC2 instances.
`aws s3 sync`	Sync a local directory with an S3 bucket.
`aws cloudformation create-stack`	Create a CloudFormation stack.

Thus ends the shortest section in the book.

Parsing JSON Output with jq

The *aws* command will often print out JSON as part of its result. For example, if you retrieve information about your DNS zones with the *aws route53 list-hosted-zones* command, you will see something similar to the following:

```
{
    "HostedZones": [
        {
            "ResourceRecordSetCount": 9,
            "CallerReference": "A036EFFA-E0CA-2AA4-813B-46565D601BAB",
            "Config": {},
            "Id": "/hostedzone/Z1Q7O2Q6MTR3M8",
            "Name": "epitech.nl."
        },
        {
            "ResourceRecordSetCount": 4,
            "CallerReference": "7456C4D0-DC03-28FE-8C4D-F85FA9E28A91",
            "Config": {},
            "Id": "/hostedzone/ZAY3AQSDINMTR",
            "Name": "awssystemadministration.com."
        }
    ]
}
```

In this example, it is trivial to find any information you might be looking for. But what if the results span multiple pages and you are interested in only a subset of the returned information? Enter *jq*. This handy tool is like *sed* for JSON data. It can be used to parse, filter and generate JSON data, and is an excellent partner to the *aws* command.

Continuing the DNS zones example, imagine we want to filter the previous list to include only the domain name:

```
aws route53 list-hosted-zones | jq '.HostedZones[].Name'
"epitech.nl."
"awssystemadministration.com."
```

In this example the output of the *aws* command is piped to *jq*. .HostedZones[].Name is a *jq* filter which acts in a similar way to CSS selectors. It parses the JSON object and returns only the Name element of each of the HostedZones.

jq can also be used to filter the results. Let us say we want to find the ResourceRecord SetCount for the epitech.nl domain:

```
aws route53 list-hosted-zones | jq '.HostedZones[] | select(.Name=="epitech.nl.").ResourceRecordSe
9
```

This example uses two filters. The first returns all of the HostedZones. This list is passed to the next filter, which uses the select() function to perform a string comparison. Finally we request the ResourceRecordSetCount element for the item that matched the string comparison.

For installation instructions and more usage examples see the *jq* homepage.

Installing the Earlier AWS Command Line Tools

Prior to *awscli*, AWS provided a number of command line tools to interact with their services and APIs. Mostly obsolete, these are still useful in some cases. Each service has its own collection of tools, which must be downloaded separately. Because the installation procedure does not vary much between packages this section will use the EC2 tools as an example. The process is essentially the same for the rest of the tools.

Unfortunately, tools cannot be found in consistent locations. This inconsistency means it is more difficult than necessary to write a script that automates the installation of these tools, especially as the URLs for some tools change with each release.

> Alestic, a great blog full of useful AWS tips, has a handy guide containing links to all of the AWS command line tools, along with shell snippets (suitable for copying and pasting) to download, extract, and install each of the packages.

By convention, it is common to store the tools in a subdirectory specific to that tool, so EC2 tools go in */usr/local/aws/ec2*, and AutoScaling tools go in */usr/local/aws/as*. The following commands create this directory, download the EC2 tools, and move the extracted files into the destination directory:

```
mkdir -p /usr/local/aws/ec2
wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
unzip ec2-api-tools.zip
mv ec2-api-tools-*/* /usr/local/aws/ec2
```

Another difference between the tools is in how they handle authentication. Some require a set of access keys, whereas others use X.509 certificates or SSH keys. The EC2 tools use access keys, which can be specified in two ways: by setting environment variables containing the access key and secret, or by using the `--aws-access-key` and `--aws-secret-key` arguments on the command line. Using environment variables is more convenient and can be more secure—because specifying the credentials as command line options means they will be visible in your shell history and the list of running processes—so I recommend you use this approach where possible.

All of the AWS command line tools require some environment variables to be set before they can be used. Set the environment variables as follows, updating the paths where necessary:

> JAVA_HOME should point to the directory used as the base when Java was installed. For example, if the output of *which java* is */usr/bin/java*, JAVA_HOME should be set to */usr*.

```
export JAVA_HOME=/usr
export EC2_HOME=/usr/local/aws/ec2
export AWS_ACCESS_KEY=your_access_key_ID
export AWS_SECRET_KEY=your_secret_access_key
export PATH=$PATH:/usr/local/aws/ec2/bin
```

After setting these variables, you can start using the command line tools, for example:

```
ec2-describe-instances   Shows information about your running instances
ec2-describe-regions     Shows the list of AWS regions
```

> By default, all AWS command line tools will operate in the US East region (us-east-1). Because US East is one of the cheapest EC2 regions this is a sensible default. You can override this behavior by setting the EC2_REGION environment variable, or passing the `--region` option on the command line.

Of course, setting these environment variables every time you wish to run the EC2 tools will quickly become tiresome, so it is convenient to set them automatically upon login. The method for achieving this will vary depending on which shell you use. If you are using Bash, for example, you will need to add them to your *$HOME/.bashrc* file. The Alestic blog post mentioned earlier includes an example *.bashrc* that sets the environment variables required for most of the tools, as well as adding each of the tool-specific directories to your PATH. Once you have installed all of the tools, your *.bashrc* might look something like this:

```
export JAVA_HOME=/usr
export EC2_HOME=/usr/local/aws/ec2
export AWS_IAM_HOME=/usr/local/aws/iam
export AWS_RDS_HOME=/usr/local/aws/rds
export AWS_ELB_HOME=/usr/local/aws/elb
export AWS_CLOUDFORMATION_HOME=/usr/local/aws/cfn
export AWS_AUTO_SCALING_HOME=/usr/local/aws/as
export CS_HOME=/usr/local/aws/cloudsearch
export AWS_CLOUDWATCH_HOME=/usr/local/aws/cloudwatch
export AWS_ELASTICACHE_HOME=/usr/local/aws/elasticache
export AWS_SNS_HOME=/usr/local/aws/sns
export AWS_ROUTE53_HOME=/usr/local/aws/route53
export AWS_CLOUDFRONT_HOME=/usr/local/aws/cloudfront

for i in $(find /usr/local/aws -type d -name bin)
do
```

```
   PATH=$i/bin:$PATH
done
PATH=/usr/local/aws/elasticbeanstalk/eb/linux/python2.7:$PATH
PATH=/usr/local/aws/elasticmapreduce:$PATH

export EC2_PRIVATE_KEY=$(echo $HOME/.aws-default/pk-*.pem)
export EC2_CERT=$(echo $HOME/.aws-default/cert-*.pem)
export AWS_CREDENTIAL_FILE=$HOME/.aws-default/aws-credential-file.txt
export ELASTIC_MAPREDUCE_CREDENTIALS=$HOME/.aws-default/aws-credentials.json

# Some tools use AWS_ACCESS_KEY, others use AWS_ACCESS_KEY_ID
export AWS_ACCESS_KEY=< your access key ID >
export AWS_SECRET_KEY=< your secret access key >
export AWS_ACCESS_KEY_ID=< your access key ID >
export AWS_SECRET_SECRET_KEY=< your secret access key >

# Change the default region if desired
#export EC2_REGION=eu-west-1
```

> Make sure you do not accidentally commit these security keys to a
> public code repository such as GitHub. There have been news re-
> ports of people scanning for accidentally-publicised AWS keys, and
> using them to gain unauthorised access to AWS accounts.

For more tools and utilities, including all of the AWS command line tools, visit the AWS
developer tools site.

First Steps With EC2 and CloudFormation

Launched in 2006, Elastic Compute Cloud (or EC2, as it is universally known) is a core part of AWS, and probably one of the better known components of the service. It allows customers to rent computing resources by the hour in the form of virtual machines (known as "instances") which run a wide range of operating systems. These instances can be customized by the user to run any software applications supported by their operating system of choice.

The idea of renting computing resources by the hour goes back to the 1960s, when it was simply not financially feasible for a company or university department to own its own dedicated computer (the idea of an individual owning their own computer seeming, at this point, to be the stuff of science fiction). This changed as computers became cheaper and more popular, and dedicated computing resources became the norm.

The explosive growth of the consumer Internet, and thus the services and applications that make up the motivation for its ever-increasing use, has helped the pendulum swing back the other way, to the point where being able to elastically increase or decrease your computing resources (and therefore costs) is a key financial advantage.

In the pre-cloud days, capacity planning required a large amount of time, planning and forward thinking. Bringing new servers online was a multi-step process with the potential for delays at every step: ordering hardware from the supplier, waiting for its arrival, visiting the datacenter to unpack and rack the server, installing and configuring the operating system and software. Renting a Virtual Private Server, while usually quicker than provisioning physical hardware, also had its own set of challenges and potential delays. With the launch of EC2, all of this was replaced with a single API call.

Particularly in the consumer web application market, it is possible for new companies to experience month after month of exponential growth. This can lead to service interruption as systems administrators struggle valiantly to ensure that the demands of their users do not surpass their supply of computing power. This process is often one of the

key factors in the success of young companies and also presents one of the most acute challenges—if you do not have enough computing capacity, your users will quickly tire of seeing error pages and move on to a competitor. But over-supply is equally terminal, as you will be paying for unused computing capacity. This contributed to the failure of many companies in the 2000 "Dot-com bubble": they spent a huge amount of money providing capacity for users who never materialized.

EC2 provides a particularly interesting approach to solving this problem. As instances can be launched and terminated automatically based on your current traffic levels, it is possible to design your infrastructure to operate at (for example) 80% utilization.

Flexibility is at the heart of the AWS product offering, and this flexibility also extends to the way one interacts with AWS. For most people, the first steps with EC2 are taken via the Web Console, which is the public face of EC2. This web application lets you control most aspects of your infrastructure, although some features (such as Auto Scaling Groups, discussed later in the book) require the use of API calls or command-line tools. Historically, Amazon has usually provided command-line tools and API access to new features before they appear in the Web Console.

At the lowest level, AWS is "simply" an HTTP-based API. You can submit a request asking for ten t1.micro instances, the API request is processed, and ten instances are launched. The Web Console is merely another way of interacting with this API.

This book will make use of all of the available methods provided by AWS. In nearly all cases the methods are interchangable. If a feature specifically requires you to use the command line tools, I will indicate this. So, if you are familiar with AWS you should feel free to ignore my recommendations and use whichever method you feel most comfortable with.

What is an Instance?

At the simplest level, an instance can be thought of as a virtual server, the same as you might rent on a monthly basic from a virtual private server (VPS) provider. Indeed, some people are using EC2 in exactly the same way as they would a VPS. While perfectly serviceable in this respect, to use it in this way ignores a number of interesting features and technologies that can make your job a lot more convenient.

Amazon Machine Images (AMIs) are the main building blocks of EC2. They allow you to configure an instance once (say, installing Apache or Nginx) and then create an image of that instance. The image can then be used to launch more instances, all of which are functionally identical to the original. Of course, some attributes—such as the IP address or instance ID—must be unique, so there will be some slight differences.

AWS Regions and Availability Zones

EC2 (and many other AWS services) operate in a number of geographic regions around the world. At the time of writing there are eight AWS Regions, each of which is further divided in to multiple *availability zones*. This geographic disparity has two main benefits: you can place your application resources close to your end users for performance reasons, and you can design your application so that it is resilient to loss of service in one particular region or availability zone. AWS provides the tools to build automatic damage control in to your infrastructure, so if an availability zone fails more resources will be provisioned in the other availability zones to handle the additional load.

Each availability zone (AZ) is located in a physically separate datacenter within its region. There are three datacenters in or around Dublin, Ireland which make up the three availability zones in the EU-West-1 region—each with separate power and network connections. In theory, this means that an outage in one AZ will not have any effect on the other AZs in the region. In practice however, an outage in one AZ can often have a knock-on effect on its neighbouring AZs, and not necessarily due to any failing on Amazon's part.

Consider a well-architected application which, in the event of an AZ failure, will distribute traffic to the remaining AZs. This will result in a number of new instances being launched in the AZs that are still available. Now consider what happens when hundreds of well-architected applications all fail-over at the same time—the rush for new instances could outstrip AWS' capability to provide them, leaving some applications with too few instances.

I should note that this is an unlikely event—although AWS has service outages like any other cloud provider, deploying your application to multiple AZs will usually be sufficient for most use cases. To sustain the loss of a significant number of AZs within a Region, applications must be deployed to multiple Regions. This is considerably more challenging than running an application in multiple AZs.

Chapter 6 will demonstrate an example application which can survive the loss of one of more AZs.

Instance Types

EC2 instances come in a range of sizes, referred to as *instance types*, to suit a number of different use cases. The instance types differ in the amount of resources allocated to them. The m1.small instance type has 1.7 GB of memory and 1 EC2 Compute Unit, whereas its significantly bigger brother m2.4xlarge has 68.4 GB of memory 26 Compute Units. In the latter case, these 26 Compute Units are spread across 8 virtual cores.

For most of the examples in the book we will use a t1.micro instance, Amazon's smallest instance type. While not very powerful, it is the cheapest available instance type which makes it ideal for our tests.

In production, picking the right instance type for each component in your stack is important to minimise costs and maximize performance, and benchmarking is key when making this decision.

Processing Power

EC2, along with the rest of AWS, is built using commodity hardware running Amazon's software to provide the services and APIs. Because Amaon adds this hardware incrementally, a number of different hardware generations are in service at any one time. When it comes to discussing the underlying hardware that makes up the EC2 cloud, Amazon plays their cards close their chest and reveals relatively little information about the exact hardware specifications. The EC2 Instance Types page describes how Amazon calculates the amount of CPU resources available to each instance type:

> "One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor."

Until Amazon made it so, this was not exactly a widely used benchmark for calculating CPU requirements, which can make it difficult to calculate which instance type you best suit your usage. Taking a scientific approach to benchmarking is the only way to really be sure you are using the right instance type.

Storage

There are three options when it comes to virtual disk storage for your instances: *instance storage* (also known as *ephemeral storage*), *Elastic Block Store* (or EBS), and solid-state drives (SSDs). All are simply block storage devices which can be attached to instances. Once attached, they can be formatted with your operating system's tools and will act like a standard disk. There are some key differences between the two types.

Instance storage is attached to the physical host that runs your instance, whereas EBS is attached over the network. This has implications in terms of disk latency and throughput, so I recommend performing another series of benchmarks to see which is best for your application.

IO speeds are not the only difference—EBS has a number of features that make it preferable to instance storage in nearly all usage scenarios. Of these, I think the most useful is the ability to create a *snapshot* from an EBS. A snapshot is a copy of an EBS volume at a particular point in time. Once you have created a snapshot you can then create additional EBS volumes that will be identical copies of the source snapshot. You could, for example, create a snapshot containing your database backups. Every time a new

instance is launched, it will have a copy of the data ready for use. EBS snapshots form the backbone of many AWS backup strategies.

When an instance is terminated, any data stored on instance storage volumes is lost permanently. EBS volumes can persist after the instance has been terminated. Given all of the additional features, I tend to recommend using EBS volumes except in a few cases, such as when you need fast temporary storage for data that can be safely lost.

Multiple volumes (of either type) can be attached to an instance, leading to pretty flexible storage configurations. It is even possible to attach multiple volumes to an instance and build a software RAID array on them—an advantage of them appearing as block storage devices to the operating system.

In June 2012 AWS began offering SSDs as a higher-performance alternative to EBS and instance storage, both of which make use of traditional "flat pieces of metal spinning really quickly" hard drive technology. SSDs initially behaved more like instance storage than EBS volumes, in that it is not possible to make a snapshot of an SSD. Data must be loaded on to the SSD each time the instance is used, which increased the amount of time it takes to bring an instance in to service. However, the massively increased IO speeds of SSD volumes can make up for this shortcoming. More recently AWS updated this offering to provide SSD-backed EBS volumes, allowing them to be snapshotted and backed up like standard EBS volumes.

Networking

At its simplest, networking in AWS is very straightforward—launching an instance with the default networking configuration will give you an instance with a public IP address, which you can immediately SSH in to. Many applications will require nothing more complicated than this. At the other end of the scale, Amazon offers more advanced solutions that can, for example, give you a secure VPN connection from your datacenter to a *Virtual Private Cloud* (VPC) within EC2.

At a minimum, an AWS instance has one network device attached—the maximum amount of network devices that can be attached depends on the instance type. Running *ifconfig* on the instance will show that it has a private IP address in the 10.0.0.0/8 range. Every instance has a *private IP* and a *public IP*—the private IP can only be used within the EC2 network.

Behind the scenes AWS will map a publicly routable IP address to this private IP, and also create two DNS entries for convenience.

For example, if an instance has a private IP of `10.32.34.116` and a public IP of `46.51.131.23`, their respective DNS entries will be `ip-10-32-34-116.eu-west-1.compute.internal` and `ec2-46-51-131-23.eu-west-1.compute.amazonaws.com`. These DNS entries are known as the *private hostname* and *public hostname*.

It is interesting to note that Amazon operates a split-view DNS system, which means it is able to provide different responses depending on the source of the request. If you query the public DNS name from outside of EC2 (i.e. not from an EC2 instance) you will receive the public IP in response. However, if you query the public DNS name from an EC2 instance in the same region, the response will contain the private IP:

```
# From an EC2 instance
mike@ip-10-32-34-116:~$ dig ec2-46-51-131-23.eu-west-1.compute.amazonaws.com +short
10.32.34.116
# From my laptop
mike$ dig ec2-46-51-131-23.eu-west-1.compute.amazonaws.com +short
46.51.131.23
```

The purpose of this is to ensure that traffic does not leave the internal EC2 network unnecessarily. This is important as AWS has a highly-granular pricing structure when it comes to networking, and they make a distinction between traffic destined for the public internet and traffic that will remain on the internal EC2 network. The full breakdown of the different cost types is available on the EC2 Pricing page.

If two instances, which are in the same availability zone, communicate using their private IPs the data transfer is free of charge. However, using the public IPs will incur "Internet transfer" charges on both sides of the connection. Although both instances are in EC2, using the public IPs means the traffic will need to leave the internal EC2 network, which will result in higher data transfer costs.

By using the private IP of your instances when possible, you can reduce your data transfer costs. AWS makes this easy with their split-view DNS system—as long as you always reference the public hostname of the instance (rather than the public IP), AWS will pick the cheapest option.

Most of the early examples in the book will use a single interface, and we will look at more exotic topologies in later chapters.

Launching Instances

The most useful thing one can do with an instance is launch it, which is a good place for us to start.

As an automation-loving sysadmin you will no doubt quickly automate this process and rarely spend much time manually launching instances. Like any task though, it is worth stepping slowly through it the first time to familiarise yourself with the process.

Launching From the Web Console

Most people take their first steps with EC2 via the Web Console, which is the public face of EC2. Our first journey through the Launch Instance Wizard will introduce a number of new concepts, so we will go through each page in the wizard and take a moment to

look at each of these new concepts in turn. Although there are faster methods of launching an instance, the wizard is certainly the best way to familiarize yourself with related concepts.

Launching a new instance of an AMI

To launch a new instance, first log in to Amazon's web console, go to the EC2 tab, and click Launch Instance. This will show the first in a series of pages which will let us configure the instance options. The first of these pages is shown in Figure 2-1.
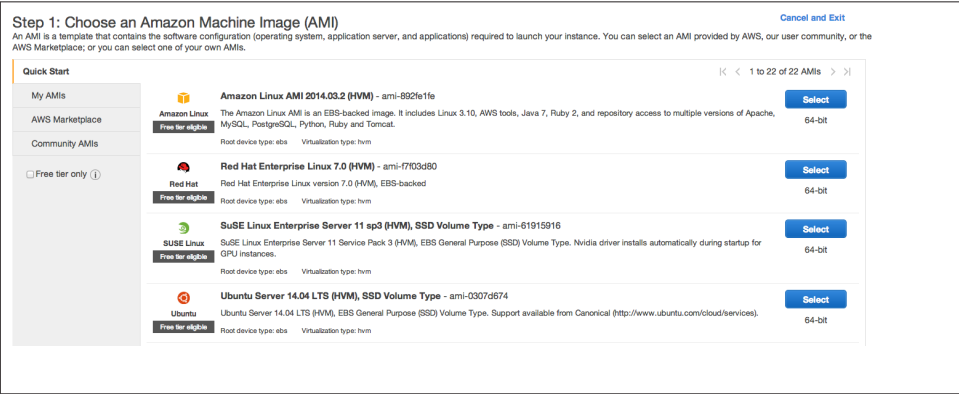
Figure 2-1. AMI Selection

As described earlier, Amazon Machine Images (AMIs) are used to launch instances that already have the required software packages installed, configured, and ready to run. Amazon themselves provide a number of AMIs for a variety of operating systems, and the Community and Marketplace AMIs provide additional choices. For example, Canonical provides officially-supported AMIs running various versions of its Ubuntu operating system. Other open-source and commercial operating systems are also available, both with and without support. The AWS Marketplace lets you use "virtual appliances" created by Amazon or third-party developers. These are Amazon Machine Images configured to run a particular set of software; for example, many people offer an AMI that runs the popular WordPress blogging software. While some of these appliances are free to use (i.e., you pay only for the underlying AWS resources you use), others require you to pay a fee on top of the cost of the Amazon resources.

If this is your first time launching an instance the My AMIs tab will be empty. Later on in this chapter we will create our own custom AMIs which will subsequently be available via this tab. The Quick Start tab lists a number of popular AMIs that are available for public use.

Click the Select button next to the Amazon Linux AMI. This gives you a number of instance types to choose from (Figure 2-2).

Figure 2-2. Selecting the instance type

EC2 instances come in a range of shapes and sizes to suit many use cases. In addition to offering increasing amounts of memory and CPU power, instance types also offer differing ratios of memory to CPU. Different components in your infrastructure will vary in their resource requirements, so it can pay to benchmark each part of your application to see which instance type is best for your particular needs.

The Micro instance class is part of Amazon's free usage tier. New customers can use 750 instance-hours free of charge with the Micro Linux and Windows instance types. After exceeding these limits, normal on-demand prices apply.

Select the checkbox next to `t2.micro` and click `Review and Launch`. Now are you presented with the review screen, which gives you a chance to confirm your options before launching the instance.

So far we have only been using the most common options when launching our instance. As you will see on the review screen, there are a number of options that we have not changed from the defaults. Some of these will be covered in great detail later in the book, whereas others will rarely be used in the most common use cases. It is worth looking through the advanced options pages to familiarise yourself with the possibilities.

User Data is an incredibly powerful feature of EC2, and one that will be used a lot later in the book to demonstrate some of the more interesting things you can do with EC2 instances. Any data entered in this box will be available to the instance once it has launched, which is a useful thing to have in your sysadmin toolbox. Among other things, user data lets you create a single AMI that can fulfill multiple roles depending on the user data it receives, which can be a huge time-saver when it comes to maintaining and updating AMIs. Some AMIs support using shell scripts as user data, so you can provide a custom script that will be executed when the instance is launched.

Furthermore, user data is accessible to configuration management tools such as Puppet or Chef, allowing dynamic configuration of the instance based on user data supplied at launch time. This is covered in further detail in Chapter 4.

The Kernel ID and RAM Disk ID options will rarely need to be changed if you are using AMIs provided by Amazon or other developers.

Termination Protection provides a small level of protection against operator error in the Web Console. When running a large number of instances, it can be easy to accidentally select the wrong instance for termination. If Termination Protection is enabled for a particular instance, you will not be able to terminate it via the Web Console or API calls. This protection can be toggled on or off while the instance is running, so there is no need to worry that you will be stuck with an immortal instance. I can personally attest to its usefulness—it once stopped me from terminating a production instance running a master database.

IAM Roles will be covered in Chapter 3. Briefly, they allow you to assign a security role to the instance. Access keys are made available to the instance so it can access other AWS APIs with a restricted set of permissions specific to its role.

Most of the time your instances will be terminated through the Web Console or API calls. Shutdown Behavior controls what happens when the instance itself initiates the shutdown, for example after running *shutdown -h now* on a Linux machine. The available options are to stop the machine so it can be restarted later, or to terminate it, in which case it is gone forever.
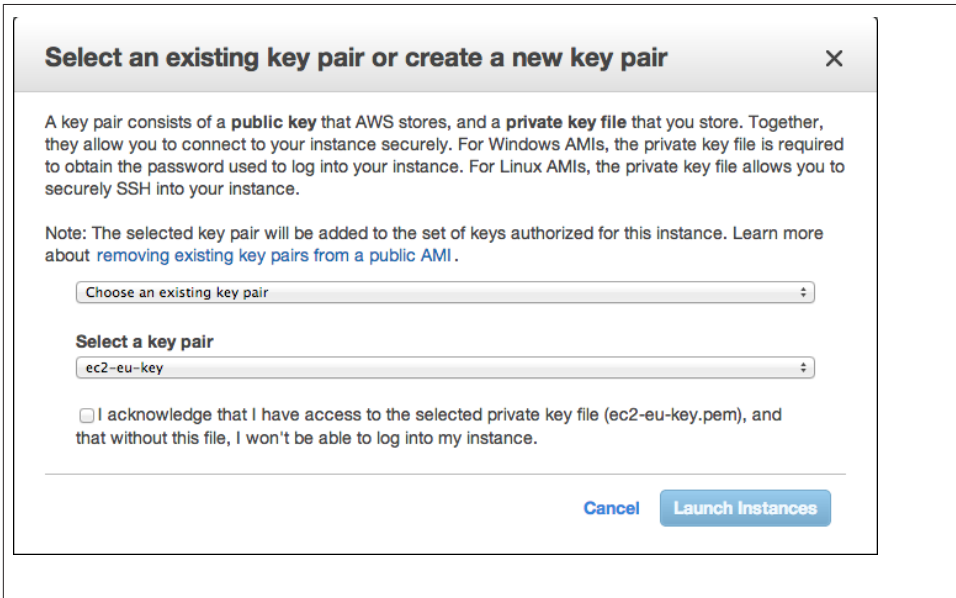
Tags are a great way to keep track of your instances and other EC2 resources via the Web Console.

If the *Cost Allocation Reports* feature (on the billing options page of your account settings page) is enabled, your CSV-formatted bill will contain additional fields allowing you to link line-item costs with resource tags.

After reviewing the options click on Launch to move to the final screen.

Key pairs

The next screen presents the available Key Pairs options (Figure 2-3).

Figure 2-3. Key pair selection

Key Pairs provide secure access to your instances. To understand the benefits of key pairs, consider how we could securely give someone access to an AMI that anyone in the world can launch an instance of. Using default passwords would be a security risk, as it is almost certain some people would forget to change the default password at some point. Amazon has thankfully implemented SSH key pairs to help avoid this eventuality. Of course, it is possible to create an AMI that uses normal usernames and passwords, but this is not the default for any AWS-supplied AMIs.

All AMIs have a "default user": when an instance is booted, the public part of your chosen key pair is copied to that user's SSH authorized keys file. This ensures you can securely log in to the instance without a password. In fact, the only thing you need to know about the instance is the default username and its IP address or hostname.

This also means that only people with access to the private part of they key pair will be able to log in to the instance. Sharing your private keys is against security best practices, so to allow others access to the instance you will need to create additional system accounts and configure them with passwords or SSH authorized keys.

The name of the default user varies between AMIs. For example, Amazon's own AMIs nearly all use `ec2user`, whereas Ubuntu's official AMIs use `ubuntu`. If you are unsure of the username, one trick you can use is to try and connect to the instance as `root`. Many AMIs will present an error message informing you that root login is disabled, and letting you know which username you should connect as.

You can create a new SSH key pair via the EC2 Key Pairs page—note that key pairs are region-specific, and this URL refers to the US-East-1 region. Keys you create in one EC2 region can not be used in another region, although you can of course upload the same to key each region instead of maintaining a specific key pair for each region.

After creating a key, a *.pem* file will be automatically downloaded. If you are using PuTTY, you will need to convert this to a PPK file using PuTTYgen before you can use it. To do this launch PuTTYgen, select Conversions → Import Key, and follow the on-screen instructions to save a new key in the correct format. Once the key has been converted it can be used with PuTTY and PuTTY Agent.

Alternatively you can upload the public part of an existing SSH key pair to AWS. This is a great help as it removes the need to add the *-i /path/to/keypair.pem* option to each SSH command.

It also means that the private part of the key pair remains entirely private—you never need to upload this to AWS, and they do not need to generate it on your behalf.

Alestic offers a handy Bash script that will import your existing public SSH key in to each region.

From the Key Pairs screen in the launch wizard, you can select which key pair will be used to access the instance, or to launch the instance without any key pair. You can select from your existing key pairs or choose to create a new key pair. It is not possible to import a new key pair at this point—if you would like to use an existing SSH key that you have not yet uploaded to AWS, you will need to upload it by following the instructions on the EC2 Key Pairs page.

Once you have created a new key pair or imported an existing one, click "Choose from your existing Key Pairs", select your key pair from the drop-down menu, and continue to the next screen. You have now completed the last step of the wizard—click Launch Instances to create the instance.

Waiting for the instance

Phew, we made it. Launching an instance can take a few minutes, depending on the instance type, current traffic levels on AWS, and a number of other factors. The Instances page of the Web Console will show you the status of your new instance. Initially this will be `pending`, while the instance is being created on the underlying physical hardware. Once the instance has been created and has begun the boot process, the page will show

the `running` state. This does not mean your instance is servicing requests or ready for you to login to, merely that the instance has been created.

Selecting an instance in the Web Console will show you its public DNS name, as well as more detail about the settings and status of the instance. At this point, you can try to SSH to the public hostname. If the connection fails, it means SSH is not yet ready to accept connections, so wait a moment and try again. Once you manage to log in to the instance, you will see a welcome screen specific to the AMI you launched.

Querying information about the instance

Now that you have an instance, what can you do with it? The answer is—anything you can do with equivalent Linux server running on physical hardware. Later chapters will demonstrate some of the more useful things you can do with EC2 instances. For now, let us take a look at the *ec2metadata* tool, which is included on many AMIs (it is available on all Linux-based AMIs from Amazon, as well as those from third parties such as Ubuntu).

The *ec2metadata* tool is very useful for quickly accessing metadata attributes of your instance: for example, the instance ID, or the ID of the AMI from which this instance was created. Running *ec2metadata* without any arguments will display all of the available metadata.

If you are interested in only one or two specific metadata attributes, you can read the values one at a time by passing the name of the attribute as a command-line option, for example:

```
mike@ip-10-32-34-116:~$ ec2metadata --instance-id
i-89580ac1
mike@ip-10-32-34-116:~$ ec2metadata --ami-id
ami-d35156a7
```

This is useful if you are writing shell scripts that need to access this information. Rather than getting all of the metadata and parsing it yourself, you can do something like this:

```
INSTANCE_ID=$(ec2metadata --instance-id)
AMI_ID=$(ec2metadata --ami-id)
echo "The instance $INSTANCE_ID was created from AMI $AMI_ID"
```


Where does the metadata come from? Every instance downloads its metadata from the following URL:

```
http://169.254.169.254/latest/meta-data/attribute_name
```

So to get the instance ID, you could request the URL *http://169.254.169.254/latest/meta-data/instance-id*.

This URL is only accessible from within the instance—if you want to query the metadata from outside the instance, you will need to use the *ec2-describe-instances* command.

Terminating the instance

Once you have finished testing and exploring the instance, you can terminate it. In the Web Console, right-click the instance and select Terminate Instance.

Next, we will look at some of other available methods of launching instances.

Launching With Command Line Tools

If you followed the steps in the previous section, you will probably have noticed a few of the drawbacks of launching instances with the Web Console. The number of steps involved and the variety of available options means documenting the process takes a lot of time to both produce and consume. This is not meant as a criticism of the Web Console—EC2 is a complicated beast, thus any interface to it requires a certain level of complexity.

Because AWS is a self-service system, it must support the use cases of many different users, each with differing requirements and levels of familiarity with AWS. By necessity the Web Console is equivalent to an enormous multi-purpose device that can print, scan, fax, photocopy, shred, and collate.

This flexibility is great when it comes to discovering and learning the AWS ecosystem, but is less useful when you have a specific task on your to-do list that must be performed as quickly as possible. Interfaces for managing production systems should be streamlined for the task at hand, and not be conducive to making mistakes.

Documentation should also be easy to use, particularly in a crisis, and the Web Console does not lend itself well to this idea. Picture yourself in the midst of a downtime situation where you need to quickly launch some instances, each with different AMIs and User Data. Would you rather have to consult a ten-page document describing which options to choose in the Launch Instance Wizard, or copy and paste some commands into the terminal?

Fortunately, Amazon gives us the tools required to do the latter. The EC2 command line tools can be used to perform any action available from the Web Console, in a fashion that is much easier to follow and more amenable to automation.

If you have not already done so, you will need to set up the EC2 command line tools according to the instructions in "Preparing Your Tools" on page 2 before continuing. Make sure you set the AWS_ACCESS_KEY and AWS_SECRET_KEY environment variables.

Access Key IDs and Secrets

When you log in to the AWS Web Console, you will usually use your email address and password to authenticate yourself. Things work a little bit differently when it comes to

the command line tools. Instead of a username and password, you use an *access key ID* and *secret access key*. Together, these are often referred to as your *access credentials*.

Although access credentials consist of a pair of keys, they are not the same as an SSH key pair. The former is used to access AWS APIs, and the latter is used to SSH in to an instance.

When you create an AWS account, a set of access credentials will be created automatically. These keys have full access to your AWS account—keep them safe! You are responsible for the cost of any resources created using these keys, so if a malicious person were to use these keys to launch some EC2 instances you would be left with the bill.

"IAM Users and Groups" on page 50 discusses how you can set up additional accounts and limit which actions they can perform. For the following examples, we will use the access keys that AWS has already created. You can find them on the Security Credentials page.

The initial versions of the AWS Command Line tools were separated based on the service with which they interacted: There was one tool for EC2, another for Elastic Load Balancers, and so on. The sheer number of available tools could be overwhelming to newcomers. Amazon has since almost entirely replaced these tools with a single unified command line tool: The AWS Command Line Interface, or awscli for short. While some services still use the "old" tools, most services can now be managed from a single application.

As a Python application awscli can be easily installed with:

```
pip install awscli
```

Once you have installed awscli you can see general usage information and a list of services that can be managed with *aws help*. For help on a specific service you can use *aws ec2 help*. Finally, help on a specific command can be displayed with *aws ec2 run-instances help*.

To launch an instance from the command line, you need to provide a number of values which correspond to the options you can choose from when using the Web Console. Because all of this information must be entered in a single command, rather than gathered through a series of web pages, it is necessary to perform some preliminary steps so you know which values to choose. The Web Console can present you with a nice dropdown box containing all of the valid AMIs for your chosen region, but to use the command line, you need to know the ID of the AMI before you can launch it.

The easiest way to get a of available images is in the Instances tab in the Web Console, which lets you search through all available AMIs. Ubuntu images can be found using Canonical's AMI Locator. Keep in mind that AMIs exist independently in EC2 regions

—the Amazon Linux AMI in the US East region is not the same image as the Amazon Linux AMI in Europe, although they are functionally identical. Amazon (and other providers) make copies of their AMIs available in each region as a convenience to their users, but the AMI will have a different ID.

If you need to find an AMI using the command-line tools, you can do so with the *aws ec2 describe-instances* command as follows:

```
# Describe all of your own images in the EU West region
aws ec2 describe-images --owners self --region eu-west-1

# Find Amazon-owned images for Windows Server 2008, 64-bit version
aws ec2 describe-images --owners amazon --filter architecture=x86_64 | grep Server-2008

# List the AMIs that have a specific set of key/value tags
aws ec2 describe-images --owners self --filter tag:role=webserver --filter tag:environment=product
```

At the time of writing the latest stable Ubuntu LTS (long-term support) version is 14.04.1. In the European EC2 region the latest version of Canonical's official AMI is ami-00b11177, which will be used in the examples. Make sure to replace this with your chosen AMI. If you are not sure which to use, or have no real preference, I recommend using the latest LTS version of Ubuntu.

The command used to launch an instance is *aws ec2 run-instances*. The most basic invocation is simply *aws ec2 run-instances* **ami-00b11177**, which will launch an m1.small instance in the us-east-1 region. However, if you run this command and attempt to log in to the instance, you will soon notice a rather large problem: because no key pair name was specified, there is no way to log in to the instance. Instead, run the command with the -key option to specify one of the SSH key pairs you created earlier. In the following example, I have also changed the instance type to t1.micro, the smallest instance:

```
mike@ip-10-32-34-116:~$ aws ec2 run-instances ami-00b11177 --region eu-west-1 --key mike --instanc
RESERVATION r-991230d1 612857642705 default
INSTANCE i-fc2067b7 ami-00b11177   pending mike 0  t1.micro 2012-11-25T15:51:45+0000 eu-west-1a ak
[ output truncated ]
```

Once EC2 receives the request to launch an instance, it prints some information about the pending instance. The value we need for the next command is the instance ID, in this case i-fc2067b7.

Although this command returns almost immediately, you will still need to wait a while before your instance is ready to accept SSH connections. You can check on the status of the instance while it is booting with the *aws ec2 describe-instance-status* command. While the instance is booting, its status will be "pending." This will change to "running" when the instance is ready. Remember that "ready" in this context means "the virtual instance has been created and the operating system's boot process has started"—it does not necessarily mean that the instance is ready for an SSH connection, which is important when writing scripts that automate these commands.

When your instance is running, the output should look similar to this:

```
mike@ip-10-32-34-116:~$ ec2-describe-instance-status --instance-ids i-fc2067b7 --region eu-west-1
INSTANCE i-fc2067b7 eu-west-1a running 16 ok ok active
SYSTEMSTATUS reachability passed
INSTANCESTATUS reachability passed
```

Another way to display information about your instance is with *aws ec2 describe-instances*, which will show much more detail. In particular it will show the public DNS name (for example, *ec2-54-247-40-225.eu-west-1.compute.amazonaws.com*) which you can use to SSH in to your instance.

```
mike@ip-10-32-34-116:~$ aws ec2 describe-instances --instance-ids i-fc2067b7 --region eu-west-1
RESERVATION r-991230d1 612857642705 default
INSTANCE i-fc2067b7 ami-00b11177 ec2-54-247-40-225.eu-west-1.compute.amazonaws.com  [ output trunc
BLOCKDEVICE /dev/sda1 vol-79b1d453 2012-11-25T15:51:49.000Z true
```

To terminate the running instance, issue *aws ec2 terminate-instance*. To verify that has instance has indeed been terminated you can use the *aws ec2 describe-instances* command again:

```
mike@ip-10-32-34-116:~$ aws ec2 terminate-instances --instance-ids i-fc2067b7 --region eu-west-1
INSTANCE i-fc2067b7 running shutting-down
mike@ip-10-32-34-116:~$ aws ec2 describe-instances --instance-ids i-fc2067b7 --region eu-west-1
RESERVATION r-991230d1 612857642705 default
INSTANCE i-fc2067b7 ami-00b11177   terminated mike 0  t1.micro 2012-11-25T15:51:45+0000 [ output t
```

As you find yourself using the command-line tools more frequently, and for more complex tasks, you will probably begin to identify procedures which are good candidates for automation. Besides saving you both time and typing, automating the more complex tasks has the additional benefits of reducing the risk of human error, and simply removing some "thinking time" from the process.

The command line tools are especially useful when it comes to documenting these procedures. Processes become more repeatable. Tasks can be more easily delegated and shared among the other members of the team.

Launching From Your Own Programs and Scripts

The command line tools are useful from an automation perspective, as it is trivial to call them from Bash or any other scripting language. While the output for some tools can be rather complex, it is relatively straightforward to parse this output and perform dynamic actions based on the current state of your infrastructure. At a certain point of complexity though, calling all of these external commands and parsing their output becomes time-consuming and error-prone. At this point, it can be useful to move to a programming language which has a client library to help work with AWS.

Officially supported client libraries are available for many programming languages and platforms, including:

- Java
- PHP
- Python
- Ruby
- .NET

Most of the examples in this book will use the Python-based Boto library, because it is the one I am most familiar with, although the other libraries are all equally capable. Even if you are not a Python developer, the examples should be easy to transfer to your language of choice, because each library is calling the same underlying AWS API.

Regardless of your language, the high-level concepts for launching an instance remain the same: first decide which attributes to will use for the instance, such as which AMI it will be created from, then issue a call to the RunInstances method of the EC2 API.

When exploring a new API from Python it can often be helpful to use the interactive interpreter. This lets you type in lines of Python code one at a time, instead of executing them all at once in a script. The benefit here is that you have a chance to explore the API and quickly get to grips with the various functions and objects that are available. I will use this method in the upcoming examples. If you prefer, you could also copy the example code to a file and run it all in one go with *python filename.py*.

If you do not already have the Boto library installed, you will need to install it with Pip (*pip install boto*) before continuing with the examples. Once this is done, open the Python interactive interpreter by running *python* without any arguments:

```
mike@ip-10-32-34-116:~$ python
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When you connect to an AWS service with Boto, Boto needs to know which credentials (i.e., which access key and secret) it should use to authenticate. You can explicitly pass the aws_access_key_id and aws_secret_access_key keyword arguments when calling connect_to_region, as shown here:

```
>>> AWS_ACCESS_KEY_ID = "your-access-key"
>>> AWS_SECRET_ACCESS_KEY = "your-secret-key"
>>> ec2_conn = connect_to_region('eu-west-1',
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY)
```

Alternatively, if the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables are set Boto will use these automatically:

```
mike@ip-10-32-34-116:~/scripts$ export AWS_SECRET_ACCESS_KEY='your access key'
mike@ip-10-32-34-116:~/scripts$ export AWS_ACCESS_KEY_ID='your secret key'
mike@ip-10-32-34-116:~/scripts$ python
Python 2.7.3 (default, Apr 20 2012, 22:39:59)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> ec2_conn = connect_to_region('eu-west-1')
```

Once you have connected to the EC2 API you can issue a call to `run_instances` to launch a new instance. You will need two pieces of information before you can do this —the ID of the AMI you would like to launch, and the name of the SSH key pair you will use when connecting to the instance.

```
>>> reservation = ec2_conn.run_instances('ami-00b11177', key_name='your-key-pair-name')
>>> instance = reservation.instances[0]
```

The call to `run_instances` does not, as might initially be suspected, return an object representing an instance. Because you can request more than one instance when calling the `run_instances` function, it returns a *reservation*, which is an object representing one or more instances. The reservation object lets you iterate over the instances. In our example we only requested one instance, so we simply take the first element of the list of instances in the reservation (in Python, that is done with `reservation.instances[0]`) to get our instance.

Now the instance is launching, and we have an instance (in the programming sense) of the instance (in the EC2 sense), we can begin to query its attributes. Some of these are available immediately, whereas others do not get set until later in the launch process. For example, the DNS name is not available until the instance is nearly running. The instance will be in the "pending" state initially. We can check on the current state by calling the `update()` function:

```
>>> instance.state
u'pending'
>>> instance.update()
u'pending'
# After some time…
>>> instance.update()
u'running'
```

Once the instance is in the `running` state we should be able to connect to it via SSH. But first we need to know its hostname or IP address, which are available as attributes on the instance object:

```
>>> instance.public_dns_name
u'ec2-54-247-155-79.eu-west-1.compute.amazonaws.com'
>>> instance.private_ip_address
u'10.246.98.144'
>>> instance.id
u'i-6abafe21'
```

Terminating a running instance is just a matter of calling the `terminate()` function. Before we do that, let's take a moment to look at how Boto can work with EC2 tags to help make administration easier. Tags can be used as a simple but effective administration database for your EC2 resources. Setting a tag is simple:

```
>>> ec2_conn.create_tags([instance.id], {'foo': 'bar'})
True
```

Once an instance has been tagged, we can use the `get_all_instances()` method to find it again. `get_all_instances()` returns a list of reservations, each of which in turn contains a list of instances. These lists can be iterated over to perform an action on all instances that match a specific tag query. As an example, we will terminate any instances that have a tag with a key of `foo` and a value of `bar`:

```
>>> tagged_reservations = ec2_conn.get_all_instances(filters={'tag:foo': 'bar'})
>>> tagged_reservations
[Reservation:r-f94361b1]
>>> tagged_reservations[0]
Reservation:r-f94361b1
>>> tagged_reservations[0].instances[0]
Instance:i-16ce8a5d
>>> for res in tagged_reservations:
...     for inst in res.instances:
...         inst.terminate()
>>>
```

Given that nearly all resource types support tagging, and Amazon provides this feature free of charge, I'm sure you can think of plenty of ways this can help you to automate and control your infrastructure. Think of it as an incredibly simple query language for your infrastructure. Conceptually, our previous example was similar to `SELECT * FROM instances WHERE tag_foo='bar'`.

The previous example iterated over all of the matching instances (only one, in this case) and terminated them. We can now check on the status of our instance, and see that it is heading towards the "terminated" state.

```
>>> instance.update()
u'shutting-down'
# After a moment or two...
>>> instance.update()
u'terminated'
```

This example only scratches the surface of what Boto (and the other client libraries) are capable of. The Boto documentation provides a more thorough introduction to other AWS services. Having the ability to dynamically control your infrastructure is one of the best features of AWS from a system administration perspective, and gives you plenty of opportunity to automate recurring processes.

Introducing CloudFormation

There is another method of launching instances that deserves its own section.

Among the Amazon Web Services features, my personal favorite is CloudFormation. It fundamentally changed how I manage my AWS infrastructures, and is something I miss terribly when working in non-AWS environments. In a nutshell, CloudFormation is a resource provisioning tool which accepts a JSON file describing the resources you require, and then creates them for you. Such a simple idea, yet so powerful.

Consider this example checklist for launching an instance. Using the three methods of launching instances we have already looked at, how could you most efficiently perform these tasks? More importantly, how would you document the process so it is repeatable?

1. Launch a t1.micro instance of ami-00b11177 in the us-east-1 region. The instance should have a 10GB EBS volume attached to the sdf device, and belong to the security group named "webservers". It should be given the string "webserver" as User Data, and have a "role" tag with the value of "webserver".

2. Create a CNAME for *www.example.com* which points to the public hostname of the instance.

If the task is a one-off procedure, it might make sense to perform it using the Web Console, but the documentation would be time-consuming to write and tedious to follow. Automating the task through programming (either by calling the EC2 command line tools, or using one of the client libraries) means the documentation could be reduced to a single command—"run this script". While benefiting the person following the documentation, this comes at a cost to whomever must write and maintain the script.

Using CloudFormation, you simply need to create a JSON-formatted file (known as a *stack template*) describing the attributes of the instance, then let AWS do the rest. The documentation is reduced to one step: "Create a stack named webservers, using the stack template webserver.json". A stack can be thought of as a collection of resources, along with a list of events associated with changes to those resources and the stack itself.

Successfully submitting a stack template to CloudFormation will result in the creation of a *stack*, which will in turn create one or more AWS resources (such as EC2 instances or Elastic Load Balancers). There are no additional scripts to write or maintain, although writing and maintaining stack templates can become rather complicated as your infrastructure grows. The CloudFormation stack template language has its own learning curve.

Being plain text files, stack templates can be stored in your revision control system alongside your application code and server configurations. The same processes used to review changes to your code can be applied to changes in your infrastructure. By browsing the history of commits to your stack templates you can quickly audit changes

to your infrastructure, as long as you are disciplined about committing changes to the repository after updating your stacks.

An additional benefit of stack templates is that they can be reused: it is possible to create multiple stacks from the same template. This can be used to give each developer their own self-contained copy of the development stack. When new members join the team, they simply need to launch a new copy of the stack and they can start familiarizing themselves with the application and infrastructure almost immediately.

The same stack template can also be used to create multiple copies of the stack in the different AWS regions. Operating an application across multiple AWS regions requires a lot of careful planning at both the application and infrastructure layers, but Cloud-Formation makes one aspect of the task very easy: by deploying a stack template to multiple regions, you can be sure that your infrastructure is identical in each region, without needing to manually configure a series of resources in each one.

Aside from the cost of the underlying resources, CloudFormation is free of charge. Although it adds a small bump in the AWS learning curve, it is well worth taking the time to deploy your infrastructure with CloudFormation, especially if you find yourself managing complicated or frequently changing infrastructures. Routing all changes to your infrastructure though a single process (i.e., updating the CloudFormation stack) is imperative when working with a team, as it gives you an easy way to answer those questions of "who changed what, and when".

For more examples of what can be achieved with CloudFormation, have a look at the example templates provided by Amazon.

Working with CloudFormation Stacks

CloudFormation stacks are themselves a type of AWS resource, and can thus be managed in similar ways. They can be created, updated and deleted via the same methods we use for interacting with other AWS services—the Web Console, command-line tools, or client libraries. They can also be tagged for ease of administration.

Creating the Stack

In this section we will start with a basic stack template which simply launches an EC2 instance. Here is one of the simplest CloudFormation stacks:

Example 2-1. Basic CloudFormation Stack in JSON

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Description" : "A simple stack that launches an instance.",
  "Resources" : {
    "Ec2Instance" : {
      "Type" : "AWS::EC2::Instance",
```

```
      "Properties" : {
        "ImageId" : "ami-00b11177",
        "InstanceType": "t1.micro"
      }
    }
  }
}
```

> CloudFormation requires stack templates to be strictly valid JSON, so keep an eye out for trailing commas when copying or modifying templates.
>
> Templates can be validated and checked for errors with the AWS command line tool, for example:
>
> ```
> aws cloudformation validate-template --template-body file://MyStack.json
> ```
>
> Some editors, including Eclipse and Vim, provide plugins to help produce and validate JSON files.

The Resources section is an object which can contain multiple children, although our example has only one (`EC2Instance`). The `EC2Instance` object has attributes which correspond to the values you can choose when launching an instance through the Web Console or command line tools.

CloudFormation stacks can be managed through the Web Console, with the command line tools, or with client libraries such as Boto.

One advantage of using the Web Console is that a list of Events is displayed in the bottom pane of the interface. With liberal use of the refresh button, this will let you know what is happening when your stack is in the "creating" or "updating" stages. Any problems encountered while creating or updating resources will also be displayed here, which makes it a good place to start when debugging CloudFormation problems. These events can also be read using the command line tools, but the Web Console output is more friendly to human parsing.

It is not possible to paste the stack template file contents into the Web Console. Rather, you must create a local text file and upload it to the Web Console when creating the stack. Alternatively, you can make the stack accessible on a web site and provide the URL instead. The same applies when using the command line tools and API.

To see the example stack in action, copy the JSON shown in Example 2-1 into a text file. You will need to substitute the AMI (ami-00b11177) with the ID of an AMI in your chosen EC2 region. Use the command line tools or Web Console to create the stack. Assuming you have configured the CloudFormation command line tools as described in "Preparing Your Tools" on page 2 and stored your stack template in a file named *example-stack.json*, you can create the stack with this command:

```
aws cloudformation create-stack --template-body file://example-stack.json --stack-name example-sta
```

If your JSON file is not correctly formed you will see a helpful message letting you know the position of the invalid portion of the file. If CloudFormation accepted the request, it is now in the process of launching an EC2 instance of your chosen AMI. You can verify this with the *aws cloudformation describe-stack-resources* and *aws cloudformation describe-stack-events* commands:

```
mike@ip-10-32-34-116:/tmp$ aws cloudformation describe-stack-events --stack-name example-stack
STACK_EVENT   example-stack   example-stack   AWS::CloudFormation::Stack   2012-11-24T16:07:17Z   CREAT
STACK_EVENT   example-stack   Ec2Instance     AWS::EC2::Instance           2012-11-24T16:07:14Z   CREAT
STACK_EVENT   example-stack   Ec2Instance     AWS::EC2::Instance           2012-11-24T16:06:40Z   CREAT
STACK_EVENT   example-stack   example-stack   AWS::CloudFormation::Stack   2012-11-24T16:06:30Z   CREAT

mike@ip-10-32-34-116:/tmp$ aws cloudformation describe-stack-resources --stack-name example-stack
STACK_RESOURCE   Ec2Instance   i-5689cc1d   AWS::EC2::Instance   2012-11-24T16:07:14Z   CREATE_COMPLETE
```

aws cloudformation describe-stack-events prints a list of events associated with the stack, in reverse chronological order. Following the chain of events, we can see that AWS first created the stack, then spent around thirty seconds creating the instance, before declaring the stack completely created. The second command shows us the ID of the newly-launched instance: `i-5689cc1d`.

Updating the Stack

Updating a running stack is equally straightforward task. But before doing this, a brief digression into the way AWS handles resource updates is called for.

Some attributes of AWS resources cannot be modified once the instance has been created. Say, for example, you launch an EC2 instance with some User Data. You then realize that the User Data was incorrect, so you would like to change it. Although the Web Console provides the option to "View/Change User Data", the instance must be in the "stopped" state before the User Data can be modified. This means you must stop the instance, wait for it to enter the stopped state, modify the User Data, and then start the instance again.

This has an interesting implication for CloudFormation. Using the previous example, imagine you have a CloudFormation stack containing an EC2 instance which has some User Data. You wish to modify the User Data, so you update the stack template file and run the *aws cloudformation update-stack* command. Because CloudFormation is unable to modify the User Data on a running instance it must instead either reload or replace the instance, depending on whether this is an EBS backed or instance-store backed instance.

If this instance was your production web server, you will have had some unhappy users during this period. Therefore, making changes to your production stacks requires some planning to ensure you won't accidentally take down your application. I won't list all of the safe and unsafe types of updates here, simply because there are so many permutations to consider that it would take an additional book to include them all. The simplest thing

is to try the operation you want to automate using the Web Console or command-line tools to find out whether they require stopping the server.

We already know that User Data can not be changed without causing the instance to be stopped, because any attempt to change the User Data of a running instance in the Web Console will fail. Conversely, we know that instance tags can be modified on a running instance via the Web Console, therefore updating instance tags with CloudFormation does not require instance replacement.

Changing some attributes, such as the AMI used for an instance, will require the instance to be replaced. CloudFormation will launch a new instance using the update AMI ID, and then terminate the old instance. Obviously, this is not something you want to do on production resources without taking care to ensure service is not disrupted while resources are being replaced. Mitigating these effects is discussed later when we look at Auto Scaling and Launch Configurations.

If in doubt, test with an example stack first. CloudFormation lets you provision your infrastructure incredibly efficiently—but it also lets you make very big mistakes with equal efficiency. With great power (which automation offers) comes great responsibility.

> Be careful when updating stacks that provide production resources. Once you submit the request to modify the stack, there is no going back. Furthermore, you can not request any additional changes until the update is complete, so if you accidentally terminate all of your production resources you will have no option but to sit back and watch it happen, after which you can begin recreating the stack as quickly as possible.
>
> To remove any doubt, review the CloudFormation documentation for the resource type you are modifying. The documentation will let you know if this resource can be updated in place, or if a replacement resource is required in order to apply changes.

To see this in action, we will first update the instance to include some tags. Update the *example-stack.json* file so that it includes the line in bold below—note the addition of the comma to the end of the proceeding line.

```
…
        "InstanceType": "t1.micro",
        "Tags": [ {"Key": "foo", "Value": "bar"}]
    }
…
```

Now we can update the running stack with *aws cloudformation update-stack*, and watch the results of the update process with *aws cloudformation describe-stack-events*.

```
mike@ip-10-32-34-116:/tmp$ aws cloudformation update-stack --template-body file://example-stack.js
arn:aws:cloudformation:eu-west-1:612857642705:stack/example-stack/e8910d40-3650-11e2-945c-5017c385
```

```
mike@ip-10-32-34-116:/tmp$ aws cloudformation describe-stack-events --stack-name example-stack
STACK_EVENT  example-stack  example-stack  AWS::CloudFormation::Stack  2012-11-24T16:42:59Z  UPDAT
STACK_EVENT  example-stack  example-stack  AWS::CloudFormation::Stack  2012-11-24T16:42:44Z  UPDAT
STACK_EVENT  example-stack  Ec2Instance    AWS::EC2::Instance          2012-11-24T16:42:43Z  UPDAT
STACK_EVENT  example-stack  Ec2Instance    AWS::EC2::Instance          2012-11-24T16:42:33Z  UPDAT
STACK_EVENT  example-stack  example-stack  AWS::CloudFormation::Stack  2012-11-24T16:42:19Z  UPDAT
STACK_EVENT  example-stack  example-stack  AWS::CloudFormation::Stack  2012-11-24T16:07:17Z  CREAT
STACK_EVENT  example-stack  Ec2Instance    AWS::EC2::Instance          2012-11-24T16:07:14Z  CREAT
STACK_EVENT  example-stack  Ec2Instance    AWS::EC2::Instance          2012-11-24T16:06:40Z  CREAT
STACK_EVENT  example-stack  example-stack  AWS::CloudFormation::Stack  2012-11-24T16:06:30Z  CREAT

mike@ip-10-32-34-116:/tmp$ cfn-describe-stack-resources --stack-name example-stack
STACK_RESOURCE  Ec2Instance  i-5689cc1d  AWS::EC2::Instance  2012-11-24T16:42:43Z  UPDATE_COMPLETE
```

Finally, the *aws ec2 describe-tags* command will show that the instance is now tagged
with "foo=bar":

```
mike@ip-10-32-34-116:/tmp$ aws ec2 describe-tags --filter "resource-type=instance" --filter "resou
TAG instance i-5689cc1d foo bar
TAG instance i-5689cc1d aws:cloudformation:stack-id arn:aws:cloudformation:eu-west-1:612857642705:
TAG instance i-5689cc1d aws:cloudformation:stack-name example-stack
TAG instance i-5689cc1d aws:cloudformation:logical-id Ec2Instance
```

Notice the additional tags in the "aws:cloudformation" namespace. When provisioning
resources which support tagging, CloudFormation will automatically apply tags to the
resource. These tags let you keep track of which stack "owns" each resource, and make
it easy to find CloudFormation-managed resources in the Web Console.

Looking Before You Leap

When you have more than one person working on a stack template, it can be easy to
find yourself in a situation where your local copy of the stack template does not match
the template used by the running stack.

Consider the scenario where two people are both making changes to a stack template
stored in a Git repository. If one person makes a change and updates the stack without
committing their changes to Git, the next person to make an update will be working
with an out-of-date stack template. The next update will then revert the previous
changes which, as discussed above, could have negative consequences. This is a typical
synchronization problem whenever you have two independent activities that should
happen in tandem: in this case, updating Git and updating the actual AWS stack.

Happily, Amazon has provided a tool which, in combination with a couple of Linux
tools, will let you be certain that your local copy of the stack does indeed match the
running version. Use the *cfn-get-template* command to get a JSON file describing the
running template, clean the output with *sed* and *head*, and finally use *diff* to compare
the local and remote versions. I did this before updating the example stack to include
tags, with the following results:

```
mike@ip-10-32-34-116:/tmp$ cfn-get-template example-stack | sed 's/TEMPLATE  "//' | head -n -1 > e
mike@ip-10-32-34-116:/tmp$ diff example-stack.running example-stack.json
9c9,10
<         "InstanceType": "t1.micro"
---
>         "InstanceType": "t1.micro",
>         "Tags": [ {"Key": "foo", "Value": "bar"}]
```

These commands could be wrapped in a simple script to save typing. Changes to production CloudFormation stacks should always be proceeded by a check like this if working in a team. This check can be incorporated into the script used for updating stacks: if it happens automatically there is no chance of forgetting it.

Deleting the Stack

Deleting a running stack will, by default, result in the termination of its associated resources. This is quite frequently the desired behavior, so makes for a sensible default, but there are times when you would like the resources to live on after the stack itself has been terminated. This is done by setting the DeletionPolicy attribute on the resource. This attribute has a default value of Delete.

All resource types also support the Retain value. Using this means that the resource will not be automatically deleted when the stack is deleted. For production resources, this can be an added safety net to ensure you don't accidentally terminate the wrong instance. The downside is that, once you have deleted the stack, you will need to manually hunt down the retained resources if you wish to delete them at a later date.

The final option for the DeletionPolicy attribute is Snapshot, which is applicable only to the subset of resources that support snapshots. At the time of writing these are RDS database instances and EBS volumes. With this value, a snapshot of the database or volume will be made when the stack is terminated.

Remember that some resources will be automatically tagged with the name of the CloudFormation stack to which they belong. This can save some time when searching for instances were created with the Retain deletion policy.

Deleting a stack is done with the *aws cloudformation delete-stack* command. Again, you can view the changes made to the stack with *aws cloudformation describe-stack-events*:

```
mike@ip-10-32-34-116:/tmp$ aws cloudformation delete-stack --stack-name example-stack

    Warning: Deleting a stack will lead to deallocation of all of the stack's
    resources. Are you sure you want to delete this stack? [Ny]y

mike@ip-10-32-34-116:/tmp$ aws cloudformation describe-stack-events --stack-name example-stack
STACK_EVENT  example-stack  Ec2Instance    AWS::EC2::Instance          2012-11-24T20:01:59Z  DELET
```

```
    STACK_EVENT  example-stack  example-stack  AWS::CloudFormation::Stack  2012-11-24T20:01:52Z  DELET
    [ output truncated ]
```

Events are available only until the stack has been deleted. You will be able to see the stack while it is in the DELETE_IN_PROGRESS state, but once it has been fully deleted, *aws cloudformation describe-stack-events* will fail.

Which Method Should I Use?

As we have already seen, AWS provides a lot of choice. When deciding which method is best for your use case, there are a number of things to consider. One of the most important is the return on investment of any effort spent automating your system administration tasks.

The main factors to consider are:

- How frequently is the action performed?
- How difficult is it?
- How many people will have to perform it?

If you are part of a small team that does not make frequent changes to your infrastructure the Web Console might be all you need. Personal preference will also play a part—some people are more at home in a web interface than they are on the command line.

Once you have a complicated infrastructure or a larger team it becomes more important that processes are documented and automated, which is not a strong point of the Web Console.

For production services, I can not recommend using CloudFormation strongly enough. Given the benefits outlined in the previous section—an audit trail, stack templates stored in source control—how could any sysadmin not immediately fall in love with this technology? Unless you have a compelling reason not to, you should be using CloudFormation for any AWS resources that are important to your infrastructure.

My golden rule for any infrastructure I am responsible for is "If it's in production, it's in git". Meaning that if a resource—application code, service configuration files, and so on—is required for that infrastructure to operate, it must be under version control. CloudFront ties in to this philosophy perfectly.

No matter how useful CloudFormation is, there will always be times where you need to perform tasks that fall outside of its capabilities. For these occasions, some combination of the command line tools and client libraries are the next best thing in terms of ease of documentation and automation.

Combining the AWS client libraries with existing system management software can be a powerful tool. Packages such as Fabric (Python) and Capistrano (Ruby) make it easy

to efficiently administer large numbers of systems. By combining these with the respective language's client library you can use them to administer a fleet of EC2 instances.

Automating too early can waste as much time as automating too late, as demonstrated in Figure 2-4. Especially at the beginning of a project, processes can change frequently and updating your automation scripts each time can be a drain on resources. For this reason I recommend using the Web Console when first learning a new AWS service— once you have performed the same task a few times you will have a clear idea of which tasks will provide the most "automation ROI".
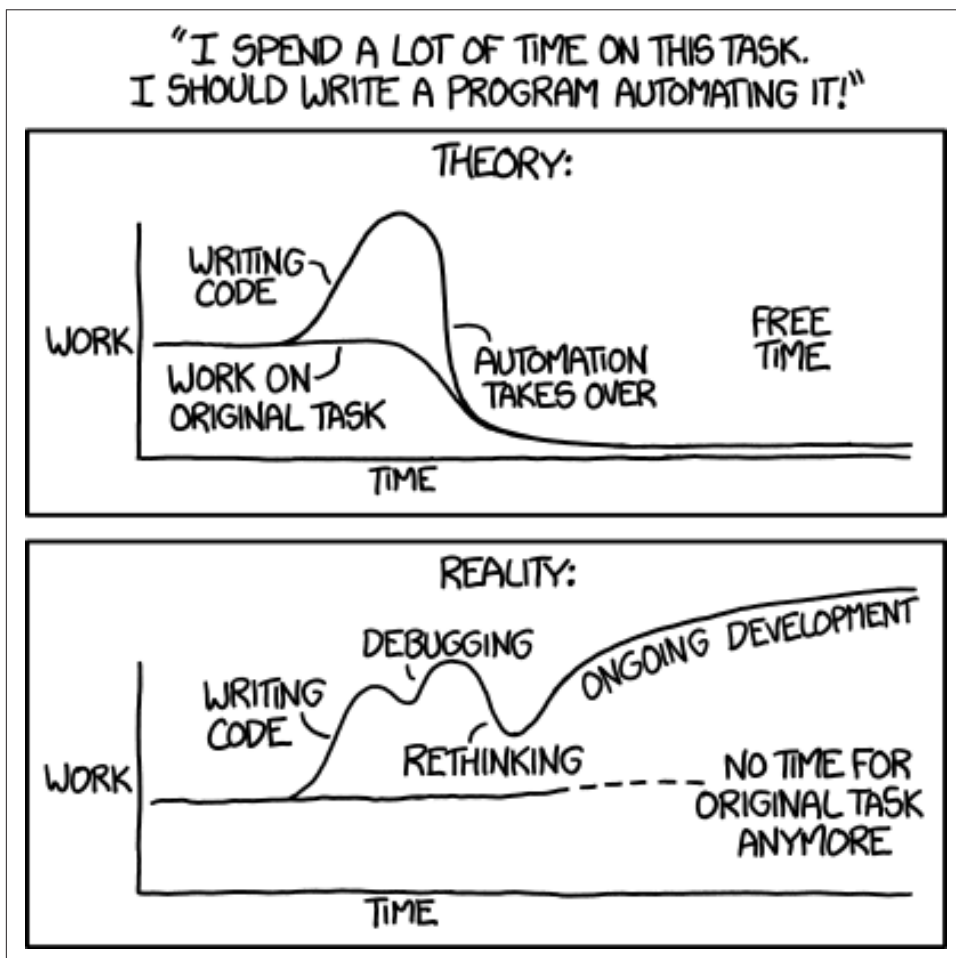
Figure 2-4. XKCD Automation

If you are interested in strategies to help decide which tasks warrant automation, I strongly recommend Time Management for System Administrators by Thomas A. Limoncelli.

Regardless of which method you choose, it is important to have a documented process describing how to perform updates. Errors will occur when there is no policy in place to inform everyone of the correct procedure.

Amazon Machine Images

AMIs are virtual appliances—pre-configured server images running a variety of operating systems and software stacks.

Amazon provide a number of their own images, running open source and commercial software, and also allows third-parties to distribute their own images through the AMI Marketplace. You can also create your own images, configured exactly to your requirements.

Building your own AMIs has a number of benefits. You get to customise the software selection and configure which services will start when the instance is launched. Any services that are not required can be disabled to cut down on wasted resources. Later chapters will show how to launch instances automatically in response to external conditions such as traffic levels (when instances are launched in response to growing demand, it is important they are ready for service as soon as possible).

Once an instance has been configured and an image created from it, that configuration is "baked in" to the AMI. As we look at configuration management tools in Chapter 4, we will see how tools like Puppet can be used to dynamically configure an instance. This raises the question of how much of the configuration should be baked in to the AMI, and how much should be dynamically configured.

At one end of the spectrum you can deploy an entirely vanilla Ubuntu image, automatically install a configuration management tool such as Puppet, and then apply your desired configuration to start up the correct services (such as Nginx for a web server). At the other end of the spectrum, you could create a custom AMI for each specific role within the application: one for the database server, one for the web server, and so on. In the latter case, all configuration options are baked in to the AMI, and no dynamic configuration is performed when the instance is launched.

In my experience the best option is somewhere in the middle, where some roles have their own AMI, whereas other AMIs perform multiple roles. The most efficient place will depend on a number of factors, including the type of software you deploy and how frequently you modify the server configuration. If it is important for newly-launched

instances to start serving requests as quickly as possible (which includes practically all uses of Auto Scaling), you'll want to reduce the amount of automatic configuration that takes place on boot.

At its core, an AMI is essentially a disk image and a metadata file describing how that disk image can be used to launch a virtual server. The metadata file keeps track of some internal information which is required when launching instances from this AMI, such as which Linux kernel to use.

In the early days of EC2, the only available AMI type was what is now known as an *instance store backed AMI*. As the Elastic Block Store service was introduced and evolved, an additional type of AMI was created: the *EBS-backed AMI*. The key architectural difference between the two is in where the disk image that contains the root volume is stored.

For EBS-backed AMIs, this is simply an EBS snapshot. When launching a new instance from such an image, a volume is created using this snapshot, and this new volume is used as the root device on the instance.

Instance store backed AMIs store their disk images in S3, which means the disk image must be copied from S3 each time an instance is launched. Because the image must be downloaded from S3 each time, the root volume size is limited to 10GB.

In practice, and EBS-backed AMI is nearly always the best option. This type of AMI can be temporarily stopped and restarted without losing any data, whereas instance store AMIs can only be terminated, at which point all data stored on the volume is lost.

Building your own AMI

AMI builds should be automated as soon as possible, if you do it with any kind of regularity. It is tedious and involves a lot of waiting around. Automating the process means you'll probably update AMIs more frequently, reducing a barrier to pushing out new features and software upgrades. Imagine you learn of a critical security flaw in your web server software which must be updated immediately. Having a procedure in place to create new AMIs and push them into production will help you respond to such scenarios rapidly and without wasting lots of time.

To demonstrate the procedure of creating an AMI, and some of the useful features that AMIs provide, let's create an AMI using the command line tools. This AMI will run an Nginx web server which displays a simple welcome page. We will look at a method of automating this procedure later in the book, in "Building AMIs With Packer" on page 108.

Begin by selecting an AMI to use as a base. I will be using an Ubuntu 14.04 image with the ID ami-00b11177. Launch an instance of this AMI with *aws ec2 run-instances*, remembering to specify a valid key pair name, then use *aws ec2 describe-instances* to find out the public DNS name for the instance.

```
mike@ip-10-32-34-116:~$ aws ec2 run-instances --image-id ami-00b11177 --region eu-west-1 --key you
RESERVATION r-991230d1 612857642705 default
INSTANCE i-fc2067b7 ami-00b11177   pending mike 0  t1.micro  [ output truncated ]
mike@ip-10-32-34-116:~$ aws ec2 describe-instances --instance-ids i-fc2067b7 --region eu-west-1
RESERVATION r-991230d1 612857642705 default
INSTANCE i-fc2067b7 ami-00b11177 ec2-54-247-40-225.eu-west-1.compute.amazonaws.com ip-10-49-118-16
```

Once the instance has launched, we need to login via SSH to install Nginx. If you are not using Ubuntu, the installation instructions might differ slightly. On Ubuntu, update the package repositories and install Nginx:

```
ubuntu@ip-10-48-62-76:~$ sudo apt-get update
ubuntu@ip-10-48-62-76:~$ sudo apt-get install nginx-full --assume-yes
```

By default Nginx is installed with a welcome page stored at */usr/share/nginx/www/ index.html*. If you like, you can modify this file to contain some custom content.

Once the instance is configured, we need to create the AMI using *ec2-create-image*. This command will automatically create an AMI from a running instance. Doing so requires that the instance be stopped and restarted, so your SSH session will be terminated when you run this command. In the background, a snapshot of the EBS volumes used for your instance will be made. This snapshot will be used when launching new instances. Because it can take some time before snapshots are ready to use, your new AMI will remain in the "pending" state for a while after *ec2-create-image* completes. The image can not be used until it enters the "available" state. You can check on the status in the Web Console, or with the *ec2-describe-images* command.

```
mike@ip-10-32-34-116:~$ aws ec2 create-image --image-id i-702c6b3b --region eu-west-1 --name test-
IMAGE ami-27a2a053
mike@ip-10-32-34-116:~$ aws ec2 describe-images --region eu-west-1 --image-ids ami-27a2a053
IMAGE ami-27a2a053 612857642705/test-image 612857642705 available private  x86_64 machine aki-6269
BLOCKDEVICEMAPPING EBS /dev/sda1  snap-8a9025a3 8 true standard
BLOCKDEVICEMAPPING EPHEMERAL /dev/sdb ephemeral0
```

When your image is ready, it can be launched by any of the means described previously. Launch a new instance based on this image, and get the public DNS name with *aws ec2 describe-instances*. If you connect via SSH, you can confirm that Nginx has started automatically:

```
ubuntu@ip-10-48-51-82:~$ service nginx status
 * nginx is running
```

Although we have configured Nginx and have a running web server, you can't access the Nginx welcome page just yet. If you try to visit the instance's public DNS name in your web browser, the request will eventually time out. This is because EC2 instances are, by default, protected by a firewall which only allows incoming SSH connections. These firewalls, known as Security Groups, are discussed in the next chapter.

Remember that both this instance and the original instance from which we created the image are still running. You might want to terminate those before moving on to the next section.

Tagging your images is a good way to keep track of them. This can be done with *aws ec2 create-tags* command. By using backticks to capture the output of shell commands, you can quickly add useful information such who created the AMI, as well as static information like the role.

```
mike@ip-10-32-34-116:~$ aws ec2 create-tags --resources ami-27a2a053 --tags created-by=`whoami`,ro
TAG image ami-65848611 created-by mike
TAG image ami-65848611 role webserver
TAG image ami-65848611 stage production
```

<div style="border:1px solid">

Tagging Strategy

Your AMI tagging strategy should let you keep track of the purpose of the image, when it was created, and its current state. Consider the lifecycle of an image: first it will be created and tested, then used in production for a while, and then finally retired when a new instance is created to replace it. The "state" tag can be used to keep track of this process, with values such as "dev", "production" or "retired". A companion "state-changed" tag can track when changes were made.

Automate the process of moving images through the lifecycle so that you never forget to add or remove the relevant tags.

</div>

Deregistering AMIs

Once an AMI is no longer required, it should be deregistered, which means it will no longer be available to use for launching new instances. Although they are not particularly expensive, it is important to regularly remove old AMIs because they clutter up the interface and contribute to a gradual increase of your AWS costs.

You must also delete the snapshot used to create the root volume—this will not happen automatically.

AWS allows you to delete the snapshot before deregistering the AMI. Doing so means you will have an AMI that looks as though it is available and ready for use, but will in fact fail when you try to launch an instance. If the deregistered AMI is referenced in Auto Scaling groups, it might be some time before you notice the problem. The only option at this point is to quickly create a new AMI and update the Auto Scaling group.

You can check to see whether a particular AMI is in use by running instances with the *aws ec2 describe-instances* command, for example:

```
mike@ip-10-32-34-116:/tmp$ aws ec2 describe-instances --filters "image-id=ami-00b11177"
RESERVATION r-b37f5cfb 612857642705 default
INSTANCE i-5689cc1d ami-00b11177   terminated  0  t1.micro [ output truncated ]
TAG instance i-5689cc1d aws:cloudformation:logical-id Ec2Instance
TAG instance i-5689cc1d foo bar
TAG instance i-5689cc1d aws:cloudformation:stack-name example-stack
TAG instance i-5689cc1d aws:cloudformation:stack-id arn:aws:cloudformation:eu-west-1:612857642705:
```

This works for individual instancse. For instances that were launched as part of an Auto
Scaling Group we can use the *aws autoscaling describe-launch-configurations* command.
Unfortunately this command does not accept a filter argument, so can not be used in
quite the same way. As a workaround you can grep the output of *aws autoscaling
describe-launch-configs* for the AMI ID.

Performing these checks before deleting AMIs en masse can save you from a rather
irritating clean up exercise.

Once you are sure the AMI is safe to deregister, you can do so with *aws ec2 deregister-
image*:

```
mike@ip-10-32-34-116:~$ aws ec2 deregister-image --image-id ami-27a2a053 --region eu-west-1
IMAGE ami-27a2a053
```

Remember to delete the snapshot that was used as the root volume of the AMI. You can
find it through the *aws ec2 describe-snapshots* command. When AWS creates a new
snapshot, it uses the description field to store the ID of the AMI it was created for, as
well as the instance and volume IDs referencing the resources it was created from. Due
to this, we can use the AMI ID as a filter in our search, returning the ID of the snapshot
we wish to delete:

```
mike@ip-10-32-34-116:~$ aws ec2 describe-snapshots --region eu-west-1 --filters "description=Creat
SNAPSHOT snap-8a9025a3 vol-c5b2d7ef completed 2012-11-25T16:16:55+0000 100% 612857642705 8 Created
mike@ip-10-32-34-116:~$ ec2-delete-snapshot --region eu-west-1 snap-8a9025a3
SNAPSHOT snap-8a9025a3
```

The same can be achieved with a simple Boto script, shown in Example 2-2. This script
will delete all images which have a `stage` tag with a value of `retired`.

Example 2-2. Deleting images with a Python script

```python
from boto.ec2 import connect_to_region

ec2_conn = connect_to_region('eu-west-1')

for image in ec2_conn.get_all_images(filters={'tag:stage': 'retired', 'tag-key': 'stage-changed'}):
    for id, device in image.block_device_mapping:
        print 'Deleting snapshot %s for image %s' % (device.snapshot_id, image.id)
        ec2_conn.delete_snapshot(device.snapshot_id)
    print 'Deleting image %s' % image.id
    ec2_conn.delete_image(image.id)
```

This script relies on your AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables being set—Boto will attempt to read these automatically. It will delete all images (and their snapshots) that have been in the "retired" stage for more than a week. To use this script, make sure your instances follow the tagging strategy described in "Tagging Strategy" on page 38. Save this file as *delete-retired-amis.py*, and use *chmod* to make it executable.

The call to `get_all_images` specifies some filter conditions: we are interested in images that have a `stage` tag with a value of `retired` and a `state-changed` tag with any value. Because we cannot use date-based filters on the server side, we must retrieve all snapshots with a "state-changed" tag, then compare the date on the client side to see whether each image is old enough for deletion.

Because deleting an image does not automatically delete the snapshot it uses for its root volume, we must do this separately by calling `delete_snapshot`.

Recap

- Benchmark your application with different instance types to make sure you are hitting the right balancing between cost and performance.
- AWS gives you plenty of choice in how you achieve your goals—use the right tools for the task at hand.
- CloudFormation is nearly always the right tool for the job.
- Automate common tasks using the command line tools or client libraries.
- Tags are essential for automation, and for keeping track of your infrastructure costs.

Access Management and Security Groups

In all of the previous examples we have been using access keys which have root-level on our AWS accounts. This means they can perform any action—including actions that potentially cost thousands of dollars in resource fees—through a few simple API calls. The thought of your AWS keys leaking should be a scary one indeed, so now is a good time to look at some of the tools Amazon provides to securely deploy and manage your applications.

Identity and Access Management

Identity and Access Management (IAM) is the name given to the suite of features that let you manage who and what can access AWS APIs using your account. This permissions based system can be somewhat overwhelming at first, but resist the temptation to give in and grant all permissions to all users. Having a well-planned policy based on IAM an important part of AWS security, and fits in well with the "defense in depth" strategy.

IAM makes a distinction between authentication ("who is this person?") and authorization ("are they allowed to perform this action?"). Authentication is handled by Users and Groups, whereas authorization is handled by IAM Policies.

Currently AWS does not provide any method of auditing user actions, although this is planned for a future release.

Amazon's CloudTrail service keeps track of the API calls made by users in your account. You can use this to review the history of AWS API calls that have been made against your account, whether they came from the Web Console, Command Line Tools or services like CloudFormation. This service is invaluable when it comes to diagnosing permissions problems.

Amazon Resource Names

You may already know that S3 bucket names must be unique across the whole of S3. Have you ever wondered how S3 bucket names must be unique, while there are surely many IAM users named "mike" or "admin"?

The answer lies with ARNs, and how they are formatted.

To identify IAM Users and other resource types, AWS uses an Amazon Resource Name (ARN). An ARN is a globally unique identifier which references AWS objects. Most AWS resource types have ARNs, including S3 buckets and IAM Users. ARNs take the following format:

arn:aws:*service*:*region*:*account_ID*:*relative_ID*

For example, here is the ARN for my IAM account (with my twelve-digit account ID replaced by Xs):

```
arn:aws:iam::XXXXXXXXXXXX:user/mike
```

Notice that the region is not specified in my user's ARN. This means that this ARN is a global resource, not tied to any specific region.

Some resources, such as S3 buckets, also omit the account ID in the ARN. S3 buckets use this ARN format:

arn:aws:s3:::*bucket_name*

For example:

```
arn:aws:s3:::mike-image-resize
```

Notice that the only variable is the bucket name. Because S3 ARNs do not include the account number, creating two S3 buckets with the same name would result in a duplicate ARN, so it is not allowed.

IAM Policies

The idea behind IAM is to separate users and groups from the actions they need to perform. You do this by creating an IAM Policy, which is a JSON-formatted document describing which actions a user can perform. This policy is then applied to users or groups, giving them access only to the services you specifically allowed.

The best way to show the flexibility of IAM policies is with an example. Let's say you use a tagging strategy described in the previous chapter, and have given all of your images a "state" tag which represents its current status, such as "production" or "retired". As a good sysadmin who dislikes repetitive tasks, you have decided to automate the process of deleting retired images—AMIs that have been replaced by newer versions, and are no longer required.

??? shows a simple Boto script that deletes any AMIs that are in the `retired` state (according to our).

This script calls a number of Boto functions, which in turn call AWS APIs to perform the requested actions. If you were to run this script, it would connect to the API using the access key and secret that are stored in your AWS_ACCESS_KEY_ID and AWS_SE-CRET_ACCESS_KEY environment variables. While convenient, those access keys have far a lot more permissions than are strictly required to run this script. Using them to authorize this script is overkill, and comes with a potential security risk—the more places in which your keys are stored, the more likely they are to be accidentally exposed to someone who does not need them.

There is another downside of reusing the same keys for multiple roles: it becomes very difficult to change them. Good security practices dictate that security credentials should be regularly rotated. If your AWS access credentials are reused in multiple scripts, keeping track of where each access key is being used becomes problematic. Replacing a key involves making the old one inactive so it can no longer be used to access APIs—if you accidentally deactivate the credentials that are used for making your database backups, you have a rather serious problem. If you do not segregate your IAM roles, you will end up being scared to deactivate old access keys because some vital component of your infrastructure will stop working.

A better solution would be to create a set of access credentials which are only authorized to perform the specific actions required by the script. Then you have a set of access credentials specific to each identifiable role—e.g. "AMI-cleaner", "database-backups", and so on.

Let's create an AMI policy with enough permissions to run the script that cleans old images and snapshots. Looking at the code, we see four Boto function calls. In most cases, Boto's functions map quite well to AWS action types. Here are the four function calls, and the action invoked by each one:

Function call	Action invoked
`connect_to_region`	`ec2:DescribeRegions`
`get_all_images`	`ec2:DescribeImages`
`delete_snapshot`	`ec2:DeleteSnapshot`
`delete_image`	`ec2:DeregisterImage`

A *permission* is a combination of two items: an *action* and one or more *resources*. AWS will check to see whether the authenticated user is allowed to perform the requested action on a specific resource—for example, is the user allowed to create a file (the action) in to an S3 bucket (the resource)?

Actions are namespaced strings which take the form *service_name:Permission*. All EC2-related permissions are prefixed with `ec2:`, such as "ec2:DeleteSnapshot".

Because policies can reference highly granular, dynamic permissions across all AWS services, they can be time-consuming to write. When you create a policy, Amazon's web interface gives you a list of permissions from which you can pick to save some time, but unfortunately no tool can magically remove the time it takes to plan out exactly which permissions each of your users or groups will require.

Using the Web Console is a great way of getting familiar with the available permissions. Even if you are a hardcore command line user, I suggest taking a few clicks around the interface to discover which actions are available. Because we already know which permissions to use for this script, we can use the command-line tools to create an user and attach a new policy to it, using the *iam-usercreate* and *iam-useraddpolicy* commands.

First we create a new user for this role, named `ami-cleaner`:

```
mike@ip-10-32-34-116:/tmp$ iam-usercreate -u ami-cleaner -k
AKIAINAK6ZGJNUAWVACA
cjJvjs79Xj/kvrJkLFpRrMAIMAxEdQrJLGIQQD28
```

The -k option says that we want to create a new access key and secret for this use. These get printed to stdout. The first item is the access key ID and the second is the secret access key. Store these somewhere safe, as we will need them later.

Next, we create an AMI policy and attach it to the user:

```
mike@ip-10-32-34-116:/tmp$ iam-useraddpolicy -u ami-cleaner -p ami-cleaner -e Allow -r "*" -a ec2:
```


Be careful with your typing: *iam-useraddpolicy* will not complain if there are typing mistakes in the action names, and will create an unusable policy.

In this case, the user and policy names are both `ami-cleaner`. The -e and -r arguments state that we are creating an "Allow" policy that applies to all resources (the asterisk after the -r option). Finally, we specify a list of actions that will be allowed, each preceded by a -a flag. You can specify as many permissions as you need.

Now we have an IAM user and policy for our role, so we can update the script to use the new keys. There are a few ways to do this, depending on your approach to managing your access keys. If no credentials are specified when opening a new connection to an AWS API Boto will check if the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables are set. Run the script by executing the following commands in a terminal:

```
export AWS_ACCESS_KEY_ID='AKIAINAK6ZGJNUAWVACA'
export AWS_SECRET_ACCESS_KEY='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'

python delete-retired-amis.py
```

The script is now being run with the most restrictive set of permissions that will still allow it to function.

By running *iam-useraddpolicy* we created a new IAM policy and added it to the user—but what does this policy actually look like? Through the IAM section in the Web Console you can view the JSON-formatted version of our new AMI. First find the "image-cleaner" user, then look in the Permissions tab.

You can also view the body of the policy with the *iam-userlistpolicies* command:

```
mike@ip-10-32-34-116:~$ iam-userlistpolicies -u ami-cleaner -v
ami-cleaner
{"Version":"2008-10-17","Statement":[{"Effect":"Allow","Action":["ec2:DescribeImages","ec2:DeleteS
IsTruncated: false
```

As you can see, the output of this command lacks formatting, which makes the Web Console a more user-friendly option if you just wish to read a policy. With formatting, our simple AMI cleaning policy looks like this:

```
{
  "Statement": [
    {
      "Action": [ A list of permission names
        "ec2:DeleteSnapshot",
        "ec2:DeregisterImage",
        "ec2:DescribeImages"
      ],
      "Effect": "Allow",
      "Resource": [ Optional qualifiers restricting which resources can be subject to these actions
        "*"
      ]
    }
  ]
}
```

Readers familiar with JSON formatting will recognize that the `Statement` attribute is actually a list. Although our example includes only one statement, a single policy can contain multiple statements. Notice that a statement can only have a single `Effect`. So to create a policy that allows some actions but denies others, we must combine multiple statements with different `Effect` attributes.

Referencing resources in IAM Policies

The Resource attribute of an IAM Policy lets you control exactly which resources an action can be performed on. In the previous example, the policy granted the user permissions to delete any EBS snapshot owned by this account. What if you want a more granular policy that applies only to a subset of resources?

As discussed previously, ARNs are used to globally identify AWS resources. Used in IAM Policies, they let you control exactly which resources are included when granting or denying permissions.

Suppose you use S3 buckets across your organization. Although most buckets contain data that could be easily replaced if lost (such as resized versions of images), you have one bucket which contains your database backups—something you certainly don't want to lose. Your users are creating and deleting buckets via the Web Console, and you would like to make sure nobody accidentally deletes your backup bucket.

The cleanest solution to this problem is to create an IAM Policy that allows users to perform any action on S3 buckets, with the exception of the one containing your backups. We do this by creating a policy containing two statements. The first grants the user all S3-related permissions, allowing them to be performed on any resource. The second statement denies all S3-related permissions, but only on the protected bucket.

When conflicting permissions are encountered, Deny takes precedence over Allow. So, the user will be able to do anything they want on any bucket, except the one containing your backups.

The policy document describing these permissions looks like this:

```
{
  "Statement": [
    { "Action": [
        "s3:*"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    },
    { "Action": [
        "s3:*"
      ],
      "Effect": "Deny",
      "Resource": [
        "arn:aws:s3:::db-backups"
      ]
    }
  ]
}
```

These examples use the "s3:*" action to grant all S3-related permissions. In practice, this is nearly always a bad idea.

When creating an IAM Policy to suit a particular role, grant the role as few permissions as possible. It might take a while (and a bit of trial and error!) to find the fewest permissions you need to fulfill a particular task, but it's worth spending the time to do so.

Resist the temptation to assign "*:*" permissions!

To implement this policy, first find the ARN of your critical bucket. Assuming your bucket is named `db-backups`, the ARN will be:

```
arn:aws:s3:::db-backups
```

When creating the policy, remember to replace the ARN I used as an example.

Next, create the policy using the command line tools or Web Console.

If using the Web Console, you can create the policy as follows:

1. Navigate to IAM Users
2. Select an existing User or Group
3. Click Attach User/Group Policy
4. Select Custom Policy
5. Paste the text into the Policy Document box

It is not currently possible to create multi-statement policies using *iam-useraddpolicy*. If you want to create the policy from the command line, you will need to write it to a temporary file and upload it with the *iam-useruploadpolicy* command. Assuming you have saved the policy in to a file named *s3-policy.json*, you can create the policy with this command:

```
iam-useruploadpolicy -u mike -p s3-policy -f s3-policy.json
```

Because denying rights to a particular activity is quite a common requirement, Amazon has provided an additional element in their resource policy language to handle this use case. Using `NotResource` as a shorthand, the above policy could be rewritten as:

```
{
  "Statement": [
    { "Action": [
        "s3:*"
```

```
      ],
      "Effect": "Allow",
      "NotResource": [
        "arn:aws:s3:::db-backups"
      ]
    }
  ]
}
```

This is almost (but not entirely—see "How Permissions are Evaluated" on page 48) the same as the longer policy we defined previously. `NotResource` refers to all objects other than the `db-backups` bucket, so this policy is effectively saying "grant all S3 permissions on all buckets except `db-backups`".

The `Action` element of a policy document also has a negated counterpart, `NotAction`. Thus, to allow a user to perform all S3 actions except `DeleteBucket`, you could include this in your policy document:

```
    "NotAction": "s3:DeleteBucket"
```

How Permissions are Evaluated

Whenever you make a request to AWS, Amazon must evaluate the request and decide whether or not it should be allowed. The logic behind this process means there is a subtle but important difference between the two IAM Policies in the previous section.

A request typically includes three bits of information: the user making the request, the action that is to be performed, and the target of the action. For example, Alice wants to terminate an EC2 instance. When receiving the request, AWS will use this information to decide whether it should be allowed by performing the following steps:

1. All policies pertaining to the user making the request are combined, including those applied by way of group membership

2. If a permission is denied by one of the policies (an "explicit deny"), the request can be immediately denied.

3. If a permission is explicitly granted by one of the policies (an "allow"), the request is allowed.

4. Finally, if the permission was not explicitly granted it is denied (a "default deny").

Given this logic, consider the differences between the policies that used `Resource` and `NotResource` as two ways to deny access to a bucket.

The first example includes an "explicit deny" that ensures no users have permissions to modify the `db-backups` S3 bucket. The second example, however, merely grants permissions to all S3 buckets *except db-backups*. There is no "explicitly deny" on the db-backups bucket in the latter case; it is handled by the "default deny".

Consequently, if the user were assigned a further policy that granted permissions to all S3 buckets, they would have permissions to delete the `db-backups` bucket and all of its contents.

Creating IAM Policies is another area where being explicit is definitely better than implicit.

For more detail on how AWS evaluates permissions, see the AWS identity and access management page describing its evaluation logic.

Dynamic policies

Conditions can be used to create dynamic IAM Policies which behave differently depending on one or more factors. The attributes of the request (such as the ARN of the requesting user, or the source IP address) can be used in Boolean expressions to control whether a request should be allowed or denied.

Some of the available attributes on which you can base your conditions are:

- Time of day
- Source IP address
- Whether the request is being made using HTTP or HTTPS

Of particular use is the `SecureTransport` attribute, which lets us check whether SSL is being used to make the request. Many of the AWS APIs can be accessed in both secure (HTTPS) and insecure (HTTP) modes. IAM Policies provide the only way to force your users to use the secure versions of these APIs.

Let's say you have an S3 bucket that is used for storing backups of confidential customer information. For regulatory reasons (or perhaps merely due to a healthy level of sysadmin paranoia) you must maintain a remote backup of this data, and the files must be transferred over an SSL connection.

This policy document would ensure that the user could not perform any actions against the `db-backups` bucket if they are connected via plain old HTTP:

```
{
    "Statement":[{
        "Effect":"Allow",
        "Action":"s3:*",
        "Resource":"arn:aws:s3:::db-backups",
        "Condition":{
            "Bool":{
                "aws:SecureTransport":"true"
            }
        }
    }
```

```
            ]
        }
```

Using Conditions, you could further enhance this policy so that:

- All connections must be secured by SSL.
- Files can only be written to the bucket by the database server (say, IP address 192.168.10.10).
- Files can only be read by the off-site backup server (say, IP address 192.168.20.20).

Of course, you could simply remember to enable the "use SSL" option of the client you use for transferring files from S3, but unless security features are enforced at a technical level they will eventually be forgotten.

> For a more thorough look at the elements of an IAM Policy, have a look at Amazon's Access Policy Lanaguage documentation.

Limitations of IAM Policies

Although powerful, IAM Policies do have some rather large drawbacks which can take some effort to work around. Chief among these is that some AWS resources do not make use of ARNs, and can therefore not be explicitly managed by IAM Policies.

EC2 instances are a good example. Because EC2 instances do not have ARNs, there is no way to reference a specific EC2 instance from an IAM Policy. Whenever you refer to EC2 permissions in a policy, the resource will be "*", which means it will apply to every instance owned by your AWS account. If a user has the `TerminateInstance` policy, he can terminate any instance, not just the ones he launched. A number of other things are simply not yet possible with IAM.

Fortunately, it is easy to see how this could be worked around with a few additions to the IAM policy language and conditional evaluation logic. I'm sure it will not be long until AWS is pleased to announce these additions.

To work around this problem now, some people have taken to operating multiple AWS accounts—one for each department. IAM Roles give you a way to securely share resources between accounts, making this an increasingly viable option.

IAM Users and Groups

Because users and groups only need to deal with authentication, they are relatively simple compared to other AWS services. If you are familiar with how Unix or Windows

handles user and group permissions, you already know the core principles behind IAM Users and Groups.

A user can be a human who logs in to the Web Console with a username and password, or a program that uses a set of access credentials to interact with AWS APIs. The user can be assigned one or more IAM Policies, which specify the actions the user is allowed to perform.

To ease administration, Users can be placed in Groups. When an IAM Policy is assigned to a Group, all members of that Group inherit the permissions designated by the IAM Policy. It is not possible to nest groups—a group can not contain other groups.

IAM is a "global" AWS service, meaning it is not tied to any particular region. An IAM User will be able to access APIs in any region, if allowed by its assigned IAM Policies.

You should create an account for each individual person that will access your account, rather than sharing the master password for your AWS account. As people leave and join the organization it will be a lot easier to revoke old security keys, and assign permissions to new accounts.

Assigning permissions to specific users has its purposes, but it is often a sign that tasks and knowledge are not being shared across the team. If Alice is the only person with `CreateSnapshot` permissions, how is Bob going to handle backups while she is on holiday?

Instead, aim to map AWS groups to specific roles within your organization, and apply the policy to the group instead. Managing updates to permissions is also a lot easier, as they will only need to be made in one place.

Organizing users and groups with paths

If you are coming from an LDAP or Active Directory background, you might be used to a little more flexibility in user and group layout. In particular, the inability to nest groups within groups can feel like a big limitation when moving from one of these systems to IAM.

Paths are an optional feature of IAM Users that can be used to implement more complicated user and group scenarios. In combination with IAM Policies, they can be used to create a hierarchical structure for your users and groups.

Suppose you have a number of departments in your organization, and you would like each department to manage their own users. No-one likes resetting passwords or setting up new accounts, so delegating this to a group of trusted users within that department will save time and headaches on all sides.

Start by creating two new groups: one to hold the normal users of the group, and one for the group admins. Let's use "dev" as an example, and create groups named `dev_admins` and `dev_users`:

```
mike@ip-10-32-34-116:~$ iam-groupcreate -g dev_admins -p "/dev"
mike@ip-10-32-34-116:~$ iam-groupcreate -g dev_users -p "/dev"
```

Next, create two users. Alice is the most responsible member of the dev team, so she will be in the `dev_admins` and `dev_users` group**s.** Bob, being slightly less responsible (or at least feigning irresponsibility to avoid being assigned additional tasks), is only in the `dev_users` group:

```
mike@ip-10-32-34-116:~$ iam-usercreate -u alice -p /dev -g dev_admins -g dev_users
mike@ip-10-32-34-116:~$ iam-usercreate -u bob -p /dev  -g dev_users
```

We can verify that the users and groups have been created with the correct paths by issuing the *iam-userlistbypath* and *iam-grouplistbypath* commands:

```
mike@ip-10-32-34-116:~$ iam-userlistbypath
arn:aws:iam::XXXXXXXXXXXX:user/dev/alice
arn:aws:iam::XXXXXXXXXXXX:user/dev/bob
mike@ip-10-32-34-116:~$ iam-grouplistbypath
arn:aws:iam::XXXXXXXXXXXX:group/dev/dev_admin
arn:aws:iam::XXXXXXXXXXXX:group/dev/dev_users
```

Now that the users and groups are set up we need to create an IAM Policy.

Notice that the ARNs for our new users and groups include "/dev" as part of the identifier. This is the magic that makes it all work. Because we can use wildcards when specifying resources in IAM policies, we can simply grant the user permission to execute IAM actions on resources that exist under the "/dev" hierarchy. As before, the asterisk indicates "all resources":

```
{
    "Statement": {
         "Effect":"Allow",
         "Action":"iam:*",
         "Resource":["arn:aws:iam::XXXXXXXXXXXX:group/dev/*",
                     "arn:aws:iam::XXXXXXXXXXXX:user/dev/*"]
    }
}
```

Save the following policy document in a text file, and upload it to AWS with the *iam-groupuploadpolicy* command (changing the file name if necessary):

```
mike@ip-10-32-34-116:~$ iam-groupuploadpolicy -g dev_admins -p dev_admins -f dev_admin.json
```

Once this policy is applied, Alice will be able to reset Bob's password or create a new user in the "/dev" hierarchy, but she will not be able to create a new user in the "/support" hierarchy.

Multi-factor authentication

Multi-factor authentication (MFA) adds an additional layer of security to your AWS account. When signing in to AWS, you will need to enter an authentication code in

---

addition to your username and password. This authentication code can be generated by a physical device or by an application running on your computer or smartphone.

Adding a second factor to the authentication process (your password being the first one) gives you a lot of protection against unauthorized account access. It is no magic bullet, but it certainly prevents a lot of common attacks that rely on a single password giving access to your account.

To the cheers of sysadmins everywhere, Amazon decided to base their multi-factor implementation (known as AWS MFA) on an open standard. The Time-Based One-Time Password Algorithm (TOTP-RFC6238) is a method of generating passwords based on a shared secret. These passwords are valid for only a short period of time, and are typically regenerated every thirty seconds or so.

Google has made multi-factor authentication an option for logging in to their services, and as a result published Google Authenticator. This is a smartphone application—available for Android, iOS and BlackBerry—which acts as a virtual multi-factor authentication device. Because it is also based on the TOTP algorithm, it works perfectly with AWS MFA, giving you a quick way to increase your AWS account security without any monetary cost.

There is of course a small time cost, as you will need to look up your access code whenever your AWS session expires. From a security perspective, it seems like a cost worth paying.

> If you have purchased a hardware multi-factor authentication device or downloaded a virtual device for your smartphone, visit the AWS Multi-Factor Authentication page to tie it into AWS.

IAM Roles

Consider the following scenario: you regularly need to resize a number of images stored in an S3 bucket. Knowing a bit about Boto, you write a script which will look in the *incoming* directory of your S3 bucket for new images, perform the resize operations, and save the resulting images in the *processed* directory.

You want to launch an EC2 instance which will run this script once it has finished booting. For the script to work, it needs to use a set of AWS credentials with permissions to read the source files, and write the output files to the S3 bucket.

In the previous section, we have already created the IAM user, applied the appropriate policy to it, and downloaded the access key and secret. But how will you provide these keys to the instance so they can be used by the log processing script?

Until June 2012, the process of distributing AWS keys to your EC2 instances was somewhat painful. There were essentially two main options: bake the keys in to the AMI so they are available when an instance boots, or provide them to the instance at runtime, perhaps with User Data.

Both had their own downsides—if keys are baked in to the AMI, replacing keys means building a new AMI. If you go for the User Data option, your unencrypted AWS keys are easily visible in the Web Console and other places. Amazon recognized that both of these options lacked security and simplicity, so they introduced IAM Roles in response.

IAM Roles almost entirely remove the problems surrounding this issue. Like Users and Groups, IAM Roles can have one or more policies applied to them. When you launch an instance, you assign it a role that you have previously created. AWS will automatically generate access credentials and make them available to the instance. These credentials can then be used to access AWS services, with the permissions specified by the role's policies.

Best of all, Amazon will regularly rotate the keys during the lifetime of the instance, without requiring any action on your part. This can be a big relief if you are working with long-running instances, as it seriously reduces the time in which compromised keys are usable.

Given these advantages, I can't recommend using IAM Roles highly enough. If all of your AWS scripts make use of roles, you will never need to worry about rotating these access credentials, even when people leave your organization. Furthermore, you will no longer run the risk of accidentally revoking keys that are still in use in some little-visited corner of your infrastructure (although you could, of course, delete an IAM policy that you need).

> If not properly configured, IAM Roles can be used for privilege escalation. Imagine a nefarious user who has permissions to launch instances with IAM Roles, but does not have permissions to delete Route 53 DNS records. By launching an instance with a role that does have these permissions, the user could easily SSH in to the instance and retrieve the credentials.
>
> IAM Policies can be used to control which roles can be assigned by a user when they launch an instance, by explicitly referencing the ARN of the role when granting the user the "iam:PassRole" permission.

To see IAM Roles in action, let's implement the example just given. To start, we will need an S3 bucket containing an example image. s3cmd is a command line tool for interacting with S3, which I have found very useful when creating S3-based backup systems. It is available in the default package repositories of many Linux systems. If you are using

Ubuntu you can install it with *apt-get install s3cmd*. Although I will use *s3cmd* in the following example, you could of course create the S3 bucket and upload an example file via the Web Console.

> Before using *s3cmd* you will need to run *s3cmd -configure*. This will write a file in your home directory containing your AWS credentials, along with some other settings.

First, create a new S3 bucket. Because S3 bucket names must be unique, you will need to choose your own name:

```
mike@ip-10-32-34-116:~$ s3cmd mb s3://mike-image-resize
Bucket 's3://mike-image-resize/' created
```

Download an example image file and copy it to the S3 bucket. I will use the O'Reilly logo as an example.

```
mike@ip-10-32-34-116:~$ wget -q http://oreilly.com/images/orn_logos/sm.ora.logo.plain.gif
mike@ip-10-32-34-116:~$ s3cmd put sm.ora.logo.plain.gif s3://mike-image-resize/incoming/
sm.ora.logo.plain.gif -> s3://mike-image-resize/incoming/sm.ora.logo.plain.gif  [1 of 1]
 1943 of 1943   100% in    0s     6.32 kB/s  done
mike@ip-10-32-34-116:~$ s3cmd ls -r s3://mike-image-resize
2012-12-02 16:48     1943   s3://mike-image-resize/incoming/sm.ora.logo.plain.gif
mike@ip-10-32-34-116:~$
```

Now that we have a bucket, we can create a policy which will allow the script to create and delete the contents of the bucket. As with most tasks involving IAM, the first step is to think about which permissions the script will require. Thankfully our example script is quite simple—the tasks it performs map to the following actions:

1. List the contents of the bucket: s3:ListBucket

2. Get the original file: s3:GetObject

3. Store the resized images in the bucket: s3:PutObject

4. Delete the processed files: s3:DeleteObject

To make our application as secure as possible, this role will have access only to the bucket we created earlier, so malicious users who managed to access these credentials would not be able to affect other parts of your application. To do this, we need to know the Amazon Resource Name (ARN) of the bucket.

As we saw near the beginning of this chapter, the ARN for an S3 bucket takes the format arn:aws:s3:::*name-of-bucket*. The consecutive colons are not merely decoration—other resource types use these fields to store additional attributes that are not used by S3 ARNs.

Because permissions can apply to either the contents of the bucket or the bucket itself, we actually need to specify two ARNs:

arn:aws:s3:::*name-of-bucket*

arn:aws:s3:::*name-of-bucket*/*

> You might consider saving some typing and simply specifying the ARN as `arn:aws:s3:::my-bucket*`. But if you have a bucket named, say, `my-bucket-secure`, you will be granting this role permissions on this bucket too. To quote the Zen of Python—"explicit is better than implicit"—even if it does sometimes involve more typing.

The first ARN references the bucket itself, and the second references any keys stored within that bucket. If we wanted to, we could assign an even more stringent set of permissions that allows the application to read and delete files in the *incoming* directory, but only write to the *processed* directory.

We now have all of the information we need to create a role and assign a policy to it. We do this with the *iam-rolecreate* command, creating a role named `image-resizing`:

```
mike@ip-10-32-34-116:~$ iam-rolecreate -r image-resizing -s ec2.amazonaws.com -v
arn:aws:iam::612857642705:role/image-resizing
AROAJ5QW2AJ237SKRFICE
{"Version":"2008-10-17","Statement":[{"Effect":"Allow","Principal":{"Service":["ec2.amazonaws.com"
```

The first line of the output is the ARN of the newly-created role; we will use this later. The `-s` flag controls which services may assume this role—at the time of writing, EC2 is the only service that can assume roles.

Once the role has been created, we can create a policy and apply it to the role:

```
mike@ip-10-32-34-116:~$ iam-roleaddpolicy -r image-resizing -p image-resizing \
-c "arn:aws:s3:::name-of-bucket" -c "arn:aws:s3:::name-of-bucket/*" \
-e Allow \
-a s3:ListBucket -a s3:GetObject -a s3:PutObject -a s3:DeleteObject
```

The role and policy are both named `image-resizing`. If you wish to distinguish between them (or just like Hungarian Notation) you might want to call the latter `policy-image-resizing`. If this command completed without any errors, the policy has been created and applied to the role.

Finally we need to create an IAM Instance Profile, which will let us launch an EC2 instance using the role:

```
mike@ip-10-32-34-116:~$ iam-instanceprofilecreate -r image-resizing -s image-resizing
arn:aws:iam::612857642705:instance-profile/image-resizing
```

To see it in action, we can now launch a new EC2 instance which assumes this role. We do this by passing the ARN of the profile—the output of the previous command—when

running *ec2-run-instances* (or by selecting from the list of IAM Roles in the Launch Instance Wizard).

To launch an instance using this role, execute the following command—remembering to update the AMI ID to reference an AMI that is available in your region, and the name of your SSH key pair. Note the last argument, which specifies the name of the instance profile we just created:

```
mike@ip-10-32-34-116:~$ ec2-run-instances ami-00b11177 --region eu-west-1 --key mike --instance-ty
```

Once the instance has booted, open an SSH session to it. In the previous section we used the *ec2metadata* tool to find display the instance's metadata. At the time of writing, *ec2metadata* does not have support for IAM roles, so we must use the common Unix *curl* command to display the credentials. The URL format for access credentials with *curl* is *http://169.254.169.254/latest/meta-data/iam/security-credentials/instance-profile-name*.

```
ubuntu@ip-10-227-45-65:~$ curl http://169.254.169.254/latest/meta-data/iam/security-credentials/im
{
  "Code" : "Success",
  "LastUpdated" : "2012-12-02T18:36:53Z",
  "Type" : "AWS-HMAC",
  "AccessKeyId" : "ASIAIYMFPJ6LTEBIUGDQ",
  "SecretAccessKey" : "gvcIA3RyERAfzDi9M8MyDEqNu0jvng2+NpXpxblr",
  "Token" : "AQoDYXdzEHwagAJFq5eT1rJcItY…",
  "Expiration" : "2012-12-03T00:54:20Z"
}
```

Fortunately, the authors of the AWS SDKs have decided to make things easy for us when using the client libraries. Boto, for example, has built-in support for IAM Roles. If you connect to an AWS service without specifying any credentials, Boto will check to see whether it is running on an EC2 instance, and whether this instance has an IAM Role. If so, Boto will use the credentials supplied by IAM to authorize your API requests.

Example 3-1 shows a simple Boto script that looks in the S3 bucket's "incoming" directory and resizes all of the files it finds. Processed files will be stored in the bucket's "processed" directory.

Example 3-1. Script accessing S3 buckets

```
#!/usr/bin/python

import tempfile
import Image
import shutil
import sys
from boto.s3.connection import S3Connection
from boto.s3.key import Key

IMAGE_SIZES = [
    (250, 250),
```

```
    (125, 125)
]

bucket_name = sys.argv[1]
# Create a temporary directory to store local files
tmpdir = tempfile.mkdtemp()
conn = S3Connection()
bucket = conn.get_bucket(bucket_name)
for key in bucket.list(prefix='incoming/'):
    filename = key.key.strip('incoming/')
    print 'Resizing %s' % filename
    # Copy the file to a local temp file
    tmpfile = '%s/%s' % (tmpdir, filename)
    key.get_contents_to_filename(tmpfile)
    # Resize the image with PIL
    orig_image = Image.open(tmpfile)
    # Find the file extension and remove it from filename
    file_ext = filename.split('.')[-1]
    for resolution in IMAGE_SIZES:
        resized_name = '%s%sx%s.%s' % (filename.rstrip(file_ext), resolution[0], resolution[1], file_e
        print 'Creating %s' % resized_name
        resized_tmpfile = '%s/%s' % (tmpdir, resized_name)
        resized_image = orig_image.resize(resolution)
        resized_image.save(resized_tmpfile)
        # Copy the resized image to the S3 bucket
        resized_key = Key(bucket)
        resized_key.key = 'processed/%s' % resized_name
        resized_key.set_contents_from_filename(resized_tmpfile)
    # Delete the original file from the bucket
    key.delete()

# Delete the temp dir
shutil.rmtree(tmpdir)
```

This script has a few dependencies, which can be installed on Ubuntu systems with:

```
sudo apt-get install gcc python-dev python-pip
sudo pip install PIL
sudo pip install --upgrade boto
```

> Make sure you are using a recent version of Boto, which has sup-
> port for IAM Roles.

Notice that the script contains no mention of access credentials—Boto will fall back to
using those provided by the IAM metadata.

Save the program to a file on the instance and execute it, passing the name of your S3 bucket as a command line argument. If everything goes to plan you should see something similar to the following:

```
ubuntu@ip-10-227-45-65:~$ python image-resizing.py your-bucket-name
Resizing sm.ora.logo.plain.gif
Creating sm.ora.logo.plain.250x250.gif
Creating sm.ora.logo.plain.125x125.gif
ubuntu@ip-10-227-45-65:~$
```

On your local computer (not the instance we just launched), you can now use *s3cmd* to show the contents of the bucket and verify that the resized images were indeed created:

```
mike@ip-10-32-34-116:~$ s3cmd ls -r s3://your-bucket-name
2012-12-02 19:42      6494   s3://mike-image-resize/processed/sm.ora.logo.plain.125x125.gif
2012-12-02 19:42     17607   s3://mike-image-resize/processed/sm.ora.logo.plain.250x250.gif
mike@ip-10-32-34-116:~$
```

Once you have finished with the example script and your own experiments, remember to terminate the instance.

> There are a number of problems with the previous example that would prevent you from using it in a high-traffic application, primarily because it would be impossible to "scale out" by launching multiple instances to process the files. Because multiple instances would be processing the bucket simultaneously, race conditions could emerge when two instances try to process the same image.
>
> If you are building something like this for production use, Simple Queue Service (SQS) would be a better bet.

By using IAM Roles, we removed the need to manually distribute and manage AWS credentials. Although there was a human behind the keyboard executing the image resizing script, it is easy to see how IAM Roles can save a lot of administration overhead, particularly when building applications based on Auto Scaling or CloudFormation.

Using IAM Roles From Other AWS accounts

In "Limitations of IAM Policies" on page 50 I mentioned that it is not possible to define an IAM policy that—for example—allows users from the Development department to launch instances and terminate them, while preventing them from terminating instances owned by the Marketing team. For most situations this limitation is not a huge problem—an instance might be accidentally terminated occasionally, but that is not the end of the world.

However, in some situations you might wish to have a strict border between your AWS resources. You might need to guarantee for regulatory purposes that members of the Support team cannot log into or terminate production instances. The only solution in

this case is to create separate AWS accounts. A nice side effect of maintaining separate AWS accounts is that you will receive a separate bill for each one, making it easy to see where your costs are going.

AWS Consolidated Billing lets you combine the bills from multiple AWS accounts, while still seeing exactly which account is responsible for each line item. This can save a lot of time in budget meetings, as arguments over who launched those m2.4xlarge instances for testing and forgot to terminate them become a thing of the past.

In November 2012, Amazon released a new feature, Cross-Account API Access, to help customers who have gone down this route. As the name suggests, Cross-Account API access provides a framework for securely sharing resources between AWS accounts. This feature is described fully in Amazon's Cross Account Access documentation.

Using IAM in CloudFormation Stacks

IAM Policies are a powerful tool on their own, but they become even more useful when combined with CloudFormation. Creating IAM Policies and Users from CloudFormation templates means that your CloudFormation stack contains everything required to run your application. The more self-contained your application is, the easier it will be to deploy and manage.

Building on the previous steps that introduced IAM Roles, we will now create a CloudFormation stack which runs the image resizing application. Everything—the S3 bucket, the IAM Policy and Role, and the EC2 instance that does the work—is contained within the stack template.

As always, a little planning is required before we start writing the CloudFormation stack template. First, consider the resources we need to create:

- IAM Role
- IAM Policy
- S3 Bucket
- EC2 instance

These resources need to reference each other—for example, the stack will create a new S3 bucket and an IAM policy that references the bucket. When CloudFormation creates resources, they are given automatically-generated names, which are not human-friendly and cannot be predicted in advance. Because of this, we must use the Ref function whenever we want to reference another resource in the stack.

Ref is an intrinsic function of CloudFormation templates. Passing it the logical name of a CloudFormation resource (i.e. the name you specify for the resource in the stack template) will return a value that can be used to reference that resource in other parts

of the same template. It can be thought of as similar to variables in programming terms: the logical name of the resource will be replaced with the actual value at runtime.

> The CloudFormation template language supports a number of built-in functions that can add a bit of flexibility to your stack templates. The full list of available functions is available on the AWS Instrinsic Functions page.
>
> This example uses the Join and Ref functions to refer to other resources in the stack. Although not quite as flexible as the domain-specific language included in tools like Chef or Puppet, this small set of functions can be combined to add some interesting features to your stack templates.

With that in mind, let us begin creating the stack template. Create a file named *image-resizing.json*, and add the preliminary boilerplate common to all templates:

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Description" : "Image resizing stack",
  "Resources" : {
```

The first resource we will define is the S3 bucket, which is the simplest:

```
    "ImageResizingBucket": {
        "Type": "AWS::S3::Bucket"
    },
```

This creates a simple S3 bucket with a logical name of `ImageResizingBucket`.

Next, we create the IAM Role, Profile and Policy:

```
    "ImageResizingRole" : {
        "Type": "AWS::IAM::Role",
        "Properties": {
            "AssumeRolePolicyDocument": {
                "Statement": [ {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": [ "ec2.amazonaws.com"]
                    },
                    "Action": ["sts:AssumeRole"]
                } ]
            },
            "Path": "/"
        }
    },
    "ImageResizingPolicies": {
        "Type": "AWS::IAM::Policy",
        "Properties": {
            "PolicyName": "root",
```

```
            "PolicyDocument": {
                "Statement": [ {
                    "Effect": "Allow",
                    "Action": [
                        "s3:ListBucket", "s3:GetObject",
                        "s3:PutObject", "s3:DeleteObject"
                    ],
                    "Resource": [
                        {"Fn::Join": [ "", [ "arn:aws:s3:::", {"Ref": "ImageResizingBucket"} ] ] }
                        {"Fn::Join": [ "", [ "arn:aws:s3:::", {"Ref": "ImageResizingBucket"}, "/*"
                    ]
                } ]
            },
            "Roles": [ {
                "Ref": "ImageResizingRole"
            } ]
        }
    },
    "ImageResizingProfile": {
        "Type": "AWS::IAM::InstanceProfile",
        "Properties": {
            "Path": "/",
            "Roles": [ {
                "Ref": "ImageResizingRole"
            } ]
        }
    },
```

The `ImageResizingRole` is an IAM Role that will be assigned to our instance. `ImageRe
sizingPolicies` contains IAM Policies (or, as in this case, a single policy) defining
which actions the use is allowed to perform. Note the use of the `Fn::Join` and `Ref`
instrinsic functions. `Ref` lets us assign a logical name for the S3 bucket (`ImageResizing
Bucket`) to an actual bucket name, such as `image-resizing-
imageresizingbucket-86q5y1qzusge`.

This value is in turn used passed to the `Fn::Join` function. `Join` combines a list of strings
in to a single string, separated by the given delimiter character. In this case, we use an
empty delimiter ("") and join two strings to create a valid ARN for the new S3 bucket.

The second use of `Fn::Join` also appends "/*" to the bucket's ARN, which is used to
declare actions that reference the bucket's contents, rather than the bucket itself.

By combining `Ref` and `Fn::Join`, we can dynamically create the ARN string used in
IAM Policies.

The `ImageResizingProfile` simply acts as a container, allowing us to assign the role to
an instance.

The next step is to declare an EC2 instance, and a Security Group that will let us SSH
in to this instance:

```
    "ImageResizingInstance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "InstanceType": "t1.micro",
        "ImageId": "ami-da1810ae",
        "KeyName": "your-ssh-key-name",
        "SecurityGroups" : [
            {"Ref": "ImageResizingSecurityGroup"}
        ],
        "IamInstanceProfile": {
            "Ref": "ImageResizingProfile"
        },
        "Tags" : [
            {"Key" : "role", "Value": "image-resizing"}
        ],
        "UserData" : {
            "Fn::Base64": {"Ref": "ImageResizingBucket"}
        }
      }
    },
    "ImageResizingSecurityGroup" : {
      "Type" : "AWS::EC2::SecurityGroup",
      "Properties" : {
        "GroupDescription" : "Allow SSH from anywhere",
        "SecurityGroupIngress" : [ {
          "IpProtocol" : "tcp",
          "FromPort" : "22",
          "ToPort" : "22",
          "CidrIp" : "0.0.0.0/0"
          }
        ]
      }
    }
```

This section creates a micro instance and assigns to it the newly-created IAM Instance Profile. It also populates the User Data with the name of the S3 bucket.

The `ImageResizingSecurityGroup` is a very simple Security Group that allows SSH access from any IP address—not the most secure of firewalls, but it will serve for this example.

Remember to update the `ImageID` and `KeyName` attributes to refer to a valid AMI and SSH Key Pair name.

The final step is to add an Outputs section:

```
    },
    "Outputs" : {
      "BucketName" : {
        "Description" : "The name of the S3 bucket",
        "Value" : { "Ref" : "ImageResizingBucket" }
      },
      "InstanceIP" : {
```

```
      "Description" : "Public IP address of the newly created EC2 instance",
      "Value" : { "Fn::GetAtt" : [ "ImageResizingInstance", "PublicIp" ] }
    }
  }
}
```

While not strictly required, Outputs can be useful especially when debugging new stacks. Outputs are visible in the Web Console, and can also be accessed from the command line with *cfn-describe-stack*. We define two outputs so we can easily see the IP address of the instance and the name of the S3 bucket.

Save all of these combined sections in to *image-resizing.json*, and create the stack with:

```
mike@ip-10-32-34-116:~$ aws cloudformation create-stack --stack-name image-resizing --template-boo
```

You can watch the progress of the stack creation in the Web Console.

> If this command fails, make sure you have set up your command line tools correctly. Also check that the IAM credentials you are using have permissions to launch new instances with IAM Roles, and create all of the resources referenced in the stack template.

Now that the instance is running, you can connect to it and run the image resizing script. Copy the script used Example 3-1 in to a file named *image-resize.py* and install the requirements listed in "IAM Roles" on page 53.

Last time we ran the script, we had to pass the bucket name as an argument. This time, we parse the bucket name from the output of the *ec2metadata* command. Alternatively, you could update the script to read the value directly from User Data instead of a command-line argument.

As before, place an example image in the *incoming/* directory of your S3 bucket, and then run the following commands to resize your test image:

```
ubuntu@ip-10-227-45-65:~$ BUCKET=$(ec2metadata --user-data)
ubuntu@ip-10-227-45-65:~$ python image-resizing.py $BUCKET
Resizing sm.ora.logo.plain.gif
Creating sm.ora.logo.plain.250x250.gif
Creating sm.ora.logo.plain.125x125.gif
```

Although we had to log in to the instance to run the script manually, it is clear that combining all of the resources required for a particular business task in to a single CloudFormation stack has benefits. Later we will look at methods of running tasks automatically when an instance boots.

Security Groups

Given the dynamic nature of EC2, which launches and terminates instances in response to changes in demand, it would be difficult to easily manage firewall rules with a traditional firewall such as iptables or pf. Defining rules when you know neither the hostname nor the IP address in advance could be tricky.

AWS provides Security Groups as an alternative (or sometimes, supplement) to normal firewall software. Security Groups consist of a series of inbound access rules. When launching an instance, one or more Security Groups are assigned to it. Their combined rulesets define which traffic is allowed to reach the instance.

Security Groups operate only on inbound network traffic, and don't provide all the features you might be used to. If you want Quality of Service or outbound access control, or if you use your firewall logs for bandwidth reporting, you will need to combine Security Groups with your own firewall software. Security Groups do, however, have some bells and whistles of their own, which we will look at in this chapter.

> When you create an AWS account, a Security Group named `de fault` will be created automatically. If you launch an instance without specifiying any Security Groups, the default Security Group will be automatically applied.
>
> It can be tempting to add your custom rules to this default group and use it for all of your instances. This leads to a maintenance and security nightmare, where removing IP addresses from the group risks breaking something in production, and leaving unknown IP addresses is somewhat risky.
>
> Having a well-planned Security Group strategy from the beginning of a project can save a lot of headaches later on.

The rules that make up a Security Group combine a source, a destination port, and a protocol. As in a traditional firewall, the source can be a single IP address (192.168.1.10) or a network block in CIDR notation (192.168.0.0/24). Using this you can define rules that allow your office IP address access to SSH on your EC2 instances, for example.

The source can also be the name of another Security Group, which is where they really begin to shine. Suppose you have a PostgreSQL server running on port 5432, which should only be accessible to a particular subset of your EC2 instances. Because instances are launched dynamically, you do not know their IP addresses in advance, so can not create rules using that method. To solve this problem, you can create Security Groups and dynamically assign instances to groups as the instances are created.

For this example, first create a Security Group. We give it a custom name, `db_clients`:

```
aws ec2 create-security-group --group-name db_clients --description "Database client security grou
```

Next, create a Securtiy Group named `db_servers`:

```
aws ec2 create-security-group --group-name db_servers --description "Database server security grou
```

Finally, create a rule in the `db_servers` group that allows members of the `db_clients` group access on TCP port 5432:

```
aws ec2 authorize-security-group-ingress --group-name db_servers --protocol tcp --port 5432 --sour
```

When launching new instances, you will need to assign the newly-created Security Groups as appropriate—i.e. PostgresSQL servers in the `db_servers` group. With this setup, you can easily ensure that all of your database clients can access PostgreSQL, while locking it down to protect it from external access.

> This method also works across AWS accounts—that is, you can reference Security Groups belonging to other AWS accounts within your own group rules. You need to prefix the Security Group name with the AWS account ID, e.g., *>123456789/group-name*.

Security Groups can also reference themselves—that is, allow members of a Security Group to access other members of that group. To see this in action, update the `db_servers` Security Group to allow itself access on TCP port 5432:

```
aws ec2 authorize-security-group-ingress --group-name db_servers --protocol tcp --port 5432 --sour
```

Now, if you have two instances in the `db_servers` group they will each be able to access the other's PostgreSQL instance—perfect for streaming replication.

I find this design pattern of role-based Security Group pairs to be a good way of controlling access to your instances. It is likely that many different types of instance will need to access your database server, such as web servers, monitoring systems, and reporting tools. Placing all of the database-access rules in a single db_servers group gives you only one thing to change if you, for example, change the port your database is running on.

Protecting Instances With SSH Whitelists

"Defense in Depth" is one of the key principles of successful IT security. SSH has an amazing track record when it comes to security, but there is no reason to let the whole world look for insecurities in your SSH setup. Security Groups can be used to limit which IP address can access SSH on your instances, creating a whitelist of trusted addresses.

Depending on your organization, this whitelist might not change frequently, and might be small enough for you to recognise each IP address. In larger teams, maintaining the

whitelist can become a chore, and it becomes difficult to know which address belongs to whom.

Implementing your SSH whitelist as a CloudFormation-managed Security Group can alleviate this headache and provide a number of other benefits in the process. First, consider the alternative—manually adding and removing addresses via the Web Console. This is undesirable for a number of reasons, chiefly that there is no audit trail. If someone accidentally deletes a rule there is no way of tracking down who did this, and reverting the change.

Maintain Strong Security Policies When Moving to the Cloud

Securely setting up your infrastructure takes time and effort. It will sometimes be an inconvenience. The trade-off between security and convenience is well-understood in the IT industry—you will need to choose the right position on this spectrum for your company's specific situation.

One common mistake is to add a rule to your default Security Group that allows SSH traffic from anywhere (0.0.0.0/0). This makes it very convenient to access your servers remotely, but will also result in your SSH port being probed twenty-four hours a day. Given how easy it is to manage SSH whitelists with Security Groups, there is no excuse for not taking the time to set it up.

I have seen people who should know better take some horribly insecure shortcuts on AWS, including the one I just mentioned. These are things that they would never consider doing on physical hardware. Just because we are on the cloud does not mean we can forget security best practices.

AWS provides a lot of tools to securely operate your infrastructure, but it does not enforce their use—that's up to your organizational policies.

The text-based nature of CloudFormation templates means we have an easy way of tracking changes to the whitelist—comitting them to a source control system such as Git when updating the list. This immediately gives us an audit trail, a change log, and an easy way to revert unwanted changes.

There is, however, one downside to managing whitelists in this way—the CloudFormation template syntax. Here is the section required to allow ingress from a single IP address:

```
"InstanceSecurityGroup" : {
  "Type" : "AWS::EC2::SecurityGroup",
  "Properties" : {
    "GroupDescription" : "A test IP address",
    "SecurityGroupIngress" : [ {
      "IpProtocol" : "tcp",
```

```
      "FromPort" : "22",
      "ToPort" : "22",
      "CidrIp" : "192.168.1.10/32"
    } ]
  }
}
```

Most of this template must be repeated for every IP address you wish to whitelist. Typing this stanza over and over again will quickly get repetive, so some people like to automate this. One common method is to have a CSV file containing IP addresses, which is used to generate the CloudFormation stack template file.

A Security Group created as part of a CloudFormation stack will have a name like *xxx*-ssh-whitelist. Resources created by CloudFormation have automatically generated names, which can make them a little difficult to reuse in other stacks—you will need to remember this name, and reference it in your other CloudFormation stacks to assign instances to this Security Group. Also, if you replace this stack (i.e. delete it and recreate it) the Security Group will have a new name. This limitation can be worked around by creating a two-stage approach to creating Security Groups.

Our current stack template performs two actions: creating the Security Group and populating it with addresses. Breaking this into two stages makes it much easier to manage Security Group whitelists with CloudFormation.

There are two ways to define which Security Group an "ingress rule" (as inbound Security Group rules are known) belongs to. In previous example above we specified a list of ingress rules as an attribute on the AWS::EC2::SecurityGroup resource type. Thus, the rules are "children" of the Security Group, so CloudFormation implicitly knows that they belong to the parent.

The other method involves creating AWS::EC2::IngressRule resources, and explicitly listing which Security Groups they belong to. So we can create the Security Group outside of CloudFormation (i.e., with the Web Console or command line tools), and then use CloudFormation to populate the list of IP addresses.

Either way, two-stage definitions give the best of both worlds. We can control which name is assigned to our Security Group, and still store the stack template in Git.

Now, you might be already jumping ahead and planning an even better Security Group layout. What about having an ssh_whitelist group that contains further Security Groups such as devs_whitelist, support_whitelist, and so on? Unfortunately, this is not supported at the moment. Security Groups cannot be nested, so this will not work as expected.

Virtual Private Networks and Security Groups

What if a whitelist is not enough? The overhead of adding and removing IP addresses from the list is not particularly gruelling, but there is an overhead. If you are frantically trying to SSH into an instance to diagnose a problem that is causing production downtime, the last thing you want is to waste time updating CloudFormation stacks before you can get to work fixing things.

Or perhaps you would like an additional layer of security in front of SSH, such as a VPN server that requires client-side certificates before allowing connections.

In these scenarios, a solution based solely on Security Groups won't quite cut it—we need a dedicated VPN server running within EC2. The VPN server acts as a *bastion host*: the secure entry point to your other instances.

This means your public servers do not need to accept SSH connections from the public Internet. Instead, their Security Groups can be configured to allow only those SSH connections that originate from the VPN instance. You no longer need to worry about script kiddies probing your ports, and there is no SSH whitelist to manage.

Because the instances will not need to receive any inbound connections from the public Internet, we can use Amazon's *Virtual Private Cloud* service in this example.

Amazon Virtual Private Cloud

Amazon Virtual Private Cloud (VPC) is a component of EC2's networking feature set that helps to improve the security of your EC2 instances. When EC2 was launched, the default behavior was to automatically assign a public IP address to each instance. Although instances were still protected by a Security Group, they were routable from the public Internet by default.

In 2009, Amazon introduced the VPC. This allowed users to create logically isolated sections in their network architectures, rather than having a single section containing all of their EC2 instances.

VPC makes it possible to create a variety of network topologies within AWS. This has benefits for people using AWS as an extension of their own datacenters, or those with specific network security requirements that can not be satisfied by Security Groups alone.

VPC introduced a number of components to EC2 that emulate features found when running your own network outside of AWS. These include subnets and routing tables, network Access Control Lists (ACLs), and the ability to specify IP address ranges for your EC2 instances and other AWS resources.

In 2013, VPC became the default for new AWS accounts. Upon using an EC2 region for the first time, a *default VPC* is automatically created, including a default subnet, routing

---

table, and other required components of a VPC. This process is essentially invisible to the user: new EC2 instances will be launched in to the default VPC automatically.

Our example will include a VPC consisting of two subnets. The first subnet will be accessible via the public Internet, and will be home to our bastion host. The second subnet will be private: it will not be routable from the Internet, and will be accessible only to the bastion host once we implement the required routing and access control rules.

As a demontration we will use the free OpenVPN as our VPN server, although the same general procedure will apply to other VPN servers as well. Luckily, the makers of OpenVPN have published an AMI that contains everything you need to run an OpenVPN server. To cut down on the installation time, we will use this ready-to-go AMI instead of installing it from scratch ourselves.

In the Security Groups section, we looked at some strategies for defining Security Groups for client/server applications such as PostgreSQL. In this section we will be creating two Security Groups:

openvpn
    This will be assigned to OpenVPN instances, and will allow the public Internet to access the OpenVPN server.

protected_ssh
    This will be assigned to all other instances, and will allow SSH access only from the OpenVPN instance(s).

These Security Groups will be created as part of the CloudFormation stacks.

The instance could be launched in any number of ways. We are going to use Cloud-Formation so we can have one stack that contains the VPN instance and Security Group, and another that contains a "protected" EC2 instance and its own Security Group.

You can find the ID of the OpenVPN AMI by searching for it in the Launch Instance wizard in the Web Console. Alternatively, OpenVPN maintains a list of AMIs for each EC2 region in the EC2 Appliance (AMI) Quick Start Guide.

At the time of writing, the ID for the OpenVPN AMI in eu-west-1 is `ami-a7c81bd0`. You will need to replace this if you are using a different region, or if a new OpenVPN AMI has since been created.

The OpenVPN AMI has two configuration phases. First, the OpenVPN installation process requires some configuration data such as the instance's hostname and the admin user's password. This takes place when the instance is booting. This configuration data can be provided as User Data or entered using the OpenVPN installer on the command line.

The second configuration phase takes place once OpenVPN is installed and running, and is done through the OpenVPN web interface. It is at this point that you can create additional users who will be allowed to access the VPN.

In this example we will perform the first configuration stage manually using the OpenVPN installer. Although using User Data is more convenient, it will leave the OpenVPN administrative account password visible in both the CloudFormation and EC2 Web Consoles, which is not desirable.

Example 3-2 shows the CloudFormation stack we will use to create the OpenVPN instance and associated resources.

Example 3-2. OpenVPN CloudFormation Stack

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Description" : "OpenVPN EC2 Instance and Security Group",

  "Parameters" : {
    "KeyName": {
      "Description" : "EC2 KeyPair name",
      "Type": "String",
      "MinLength": "1",
      "MaxLength": "255",
      "AllowedPattern" : "[\\x20-\\x7E]*",
      "ConstraintDescription" : "can contain only ASCII characters."
    },
    "AllowedIPRange" : {
      "Description" : "IP Range allowed to access OpenVPN via SSH and HTTP(S)",
      "Type": "String",
      "MinLength": "9",
      "MaxLength": "18",
      "Default": "0.0.0.0/0",
      "AllowedPattern": "(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})/(\\d{1,2})",
      "ConstraintDescription": "Must be a valid IP CIDR range of the form x.x.x.x/x."
    },
    "AMI" : {
      "Description" : "OpenVPN AMI ID",
      "Type": "String"
    }
  },

  "Resources" : {
    "OpenVPNInstance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
       "InstanceType" : "t2.micro",
        "SecurityGroups" : [ { "Ref" : "OpenVPNSecurityGroup" } ],
        "KeyName" : { "Ref" : "KeyName" },
        "ImageId" : { "Ref" : "AMI"},
        "SourceDestCheck" : "false"
      }
```

```
    },

    "OpenVPNSecurityGroup" : {
      "Type" : "AWS::EC2::SecurityGroup",
      "Properties" : {
        "GroupDescription" : "Allow SSH, HTTPS and OpenVPN access",
        "SecurityGroupIngress" : [        {              "IpProtocol" : "tcp",        "FromPort" : "
        ]
      }
    }
  },

  "Outputs" : {
    "InstanceId" : {
      "Description" : "InstanceId of the OpenVPN EC2 instance",
      "Value" : { "Ref" : "OpenVPNInstance" }
    },
    "OpenVPNSecurityGroup" : {
      "Description" : "ID of the OpenVPN Security Group",
      "Value" : { "Fn::GetAtt" : [ "OpenVPNSecurityGroup", "GroupId" ] }

    },
    "PublicIP" : {
      "Description" : "Public IP address of the newly created EC2 instance",
      "Value" : { "Fn::GetAtt" : [ "OpenVPNInstance", "PublicIp" ] }
    }
  }
}
```

Save this stack template to a file named *openvpn.json*, and create the stack with the CloudFormation command-line tools:

```
aws cloudformation create-stack --stack-name openvpn-server \
  --template-body file://openvpn_cloudformation.json \
  --region=eu-west-1 --parameters ParameterKey=KeyName,ParameterValue=your-key-name \
  ParameterKey=AllowedIPRange,ParameterValue=0.0.0.0/0 \
  ParameterKey=AMI,ParameterValue=ami-a7c81bd0
```

The Parameters required by the stack are specified on the command line with the syntax ParameterKey=KeyName,ParameterValue=*value*. You will need to replace the SSH key name and IP range with your own values.

If you are not launching the stack in the eu-west-1 region, you will also need to change the AMI parameter to match the OpenVPN AMI ID for your region. You can find the ID on the OpenVPN AWS AMI Marketplace page.

Some Marketplace AMIs—including the OpenVPN image—require you to accept a license agreement before the instance can be launched. To do this, go to the Marketplace page for the AMI and manually launch an instance. Until you have done this you will not be able to launch an instance of this AMI through Cloudformation.

After the stack has been created, we can find the IP address of the instance by querying its Outputs:

```
aws cloudformation describe-stacks --stack-name openvpn-server
```

This command will output a full description of the stack, including the Outputs and their values. Instead of searching through this information manually, you could use the *jq* tool to filter the JSON and print only the required values. We can use the following filter:

```
aws cloudformation describe-stacks --stack-name openvpn-server | \
    jq '.Stacks[0].Outputs[] | \
    select(.OutputKey=="OpenVPNSecurityGroup" or .OutputKey=="PublicIP").OutputValue'
```

This command will parse the JSON and print the OutputValue for the `OpenVPNSecurityGroup` and `PublicIP` Outputs, for example:

```
"sg-04822561"
"54.77.169.158"
```

Now that the instance is running, it must be configured. Begin by connecting to the instance via SSH as the `openvpnas` user. After logging the connection, an OpenVPN configuration process will be automatically started. The default choices presented by the application are suitable for our uses, so press the Enter key on each line to accept them.

Once this process exits you will need to set the password for the `openvpn` user, used to configure OpenVPN through its web interface. Generate a password, and set it by executing:

```
sudo passwd openvpn
```

You can now open the configuration page in your web browser. The address will be displayed once the OpenVPN configuration process completes, and will be something like `https://54.77.153.76:943/admin`.

When you open this address in your web browser, you should see the OpenVPN welcome page. Using the OpenVPN Quick Start Guide, you can now configure the OpenVPN server according to your requirements.

It is possible to create DNS records from CloudFormation templates, so we could in fact set up a CNAME so we can access this instance by visiting, for example, `vpn.example.com`.

After the VPN server has been configured you should now be able to connect to the VPN, using the OpenVPN documentation for your platform of choice.

This OpenVPN sever is a single point of failure, which is not desirable when it is the only way you can SSH in to your other instances. Before using a solution like this in production, you should explore methods of making this system more robust. For example you could run an Auto Scaling Group with one or more instances OpenVPN so that failed instances are automatically replaced.

Now that the VPN server is working, we can verify that it is working as expected when it comes to protected our instances. We will do this by launching a new instance and assigning it to the `protected_ssh` Security Group. Example 3-3 shows a simple Cloud-Formation stack template that declares a single instance using this Security Group.

Example 3-3. Launching a Protected Instance with CloudFormation

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Description" : "Example EC2 instance behind an OpenVPN server",

  "Parameters" : {
    "KeyName": {
      "Description" : "EC2 KeyPair name",
      "Type": "String",
      "MinLength": "1",
      "MaxLength": "255",
      "AllowedPattern" : "[\\x20-\\x7E]*",
      "ConstraintDescription" : "can contain only ASCII characters."
    },
    "AMI" : {
      "Description" : "AMI ID",
      "Type": "String"
    },
    "OpenVPNSecurityGroup" : {
      "Description" : "OpenVPN Security Group ID",
      "Type": "String"
    }
  },

  "Resources" : {
    "Ec2Instance" : {
```

```
        "Type" : "AWS::EC2::Instance",
        "Properties" : {
          "InstanceType" : "t1.micro",
          "SecurityGroups" : [ { "Ref" : "InstanceSecurityGroup" } ],
          "KeyName" : { "Ref" : "KeyName" },
          "ImageId" : { "Ref" : "AMI"}
        }
      },

      "InstanceSecurityGroup" : {
        "Type" : "AWS::EC2::SecurityGroup",
        "Properties" : {
          "GroupDescription" : "Allows SSH access from the OpenVPN instance",
          "SecurityGroupIngress" : [          {               "IpProtocol" : "tcp",          "FromPort" : "
          ]
        }
      }
    },

    "Outputs" : {
      "PrivateIP" : {
        "Description" : "Private IP address of the EC2 instance",
        "Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PrivateIp" ] }
      },
      "PublicIP" : {
        "Description" : "Public IP address of the  EC2 instance",
        "Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PublicIp" ] }
      }
    }
}
```

Save this file as *protected_instance_cloudformation.json* and execute the following command to create the stack:

```
aws cloudformation create-stack --stack-name openvpn-test-instance \
    --template-body file://protected_instance_cloudformation.json \
    --region=eu-west-1 --parameters ParameterKey=KeyName,ParameterValue=mike-ryan \
    ParameterKey=AMI,ParameterValue=ami-672ce210
    ParameterKey=OpenVPNSecurityGroup,ParameterValue=sg-04822561
```

As before, you will need to adjust some the parameters to match your environment. The value for the OpenVPNSecurityGroup should be the value retrieved from the *describe-stacks* command executed earlier.

Find out the public and private IPs of the instance by running:

```
aws cloudformation describe-stacks --stack-name openvpn-test-instance | jq '.Stacks[0].Outputs[]'
```

Once the stack has been created, make sure your VPN is disconnected and try to SSH to the public IP of the instance. This should time out, because your public IP address is not allowed to access instances in this Security Group.

Connect to the VPN, and then try to SSH to the instance's private IP address. This time you should be presented with the familiar Ubuntu prompt, confirming that our Security Groups are doing their job and traffic is being routed over the VPN.

Setting up the OpenVPN server, and performing on-going maintenance, adds overhead that is not present when working with Security Groups on their own. However, for some companies, the additional layer of security is a must. Managing this stack with Cloud-Formation will keep the maintenance to a miniumum.

This is just an introduction to running OpenVPN with CloudFormation. TurnKey Linux has pubilshed an example CloudFormation template that includes a VPN and separate subnets for the OpenVPN server and protected instances. Available on their Git-Hub page, this is a great starting point for building highly-available and secure VPNs in Amazon's cloud.

Recap

- Most of the security concepts you already know have direct parallels in AWS.
- Manually maintaining Security Groups and whitelists is time-consuming and error prone—don't do it. Automate these processes using CloudFormation or your own custom scripts.
- It is a lot easier to build a secure infrastructure from the beginning than it is to improve security on a runnning infrastructure.
- Time spent defining a sensible IAM policy at the beginning of a project will pay off in reduced headaches later in the project.
- The cloud is not a magic place where security rules do not apply. Just as your datacenter managers would not prevent you from deploying a physical server with a misconfigured firewall, AWS will not prevent you from building an insecure infrastructure.
- A VPN-based bastion host can add an additional security and auditing layer, at the cost of increased maintenance.
- Allowing SSH from 0.0.0.0/0 is nearly always a bad idea.

Configuration Management

Why Use Configuration Management?

I originally became enamoured (obsessed might be a better word) with the idea of automatically configuring and deploying infrastructures after reading the classic Bootstrapping an Infrastructure paper from the LISA '98 system administration conference.

The paper described the experiences of systems administrators and architects responsible for building and maintaining large Unix environments for financial trading-floors, and outlines some philosophies they adopted as a result. While somewhat dated in terms of the software used, the underlying princples are still highly relevant to managing today's cloud infrastructures.

"We recognize that there really is no "standard" way to assemble or manage large infrastructures of UNIX machines. While the components that make up a typical infrastructure are generally well-known, professional infrastructure architects tend to use those components in radically different ways to accomplish the same ends. In the process, we usually write a great deal of code to glue those components together, duplicating each others' work in incompatible ways. Because infrastructures are usually ad hoc, setting up a new infrastructure or attempting to harness an existing unruly infrastructure can be bewildering for new sysadmins. The sequence of steps needed to develop a comprehensive infrastructure is relatively straightforward, but the discovery of that sequence can be time-consuming and fraught with error. Moreover, mistakes made in the early stages of setup or migration can be difficult to remove for the lifetime of the infrastructure."

The authors of this paper recognised that automation is the key to effective system administration, and that a huge amount of time was being wasted by duplicating efforts to automate common tasks like installing software packages. By describing some battletested experiences they hoped to reduce the amount of redundant work performed by their peers.

Configuration management software evolved out of a wider need to address this problem. Puppet, Chef and Ansible are just a few examples of configuration management packages. They provide a framework for describing your application/server configuration in a text-based format. Instead of manually installing Apache on each of your web servers, you can instead write a configuration file which says "all web servers must have Apache installed".

As well as reducing manual labour, storing your configurations as text files means you can store them in a source control system with your application code so changes can be audited and reverted.

Your infrastructure effectively becomes "self-documenting", as your server and application configurations can be reviewed at any time by browsing your source control system.

Finally, the entire unit is a self-contained unit that can be deployed with minimal human interaction. Once the first server, running the configuration management server, is running, the rest of the infrastructure can be brought in to service without having to manually configure operating systems or applications.

This is especially important when a small team is responsible for a large infrastructure, or a in the case of consulting companies, a number of disparate infrastructures. Manually installing software does not scale up very well—if it takes you one hour to configure a server manually, it will take you two hours to configure two servers (assuming the tasks are not run in parallel).

However, if it takes one hour to configure a server with configuration management software, that configuration can be reused for as many servers as you need.

Adopting the configuration management philosophy does involve an initial time investment if you have not used it before, but it will soon pay off by reducing the amount of time you spent configuring servers and deploying changes.

OpsWorks

Amazon recognises how important configuration management tools are, and is doing their bit to make these tools more effective when working within AWS. In February 2013 they announced the OpsWorks service, bringing joy to the hearts of sysadmins everywhere.

OpsWorks makes configuration management a core part of AWS, bringing support for Chef directly in to the Web Console. It works by letting the users define the "layers" that make up their application—for example, clusters of web and database servers would be two separate layers. These layers consist of EC2 instances (or other AWS resources) that have been configured using Chef recipes. Once your layers have been defined, AWS will take care of provisioning all of the required resources.

Your running application—and all of its layers—are visible in the Web Console, making it easy to see the overall health of your application.

This makes it a lot easier for people who are familiar with Chef to use it to manage their AWS infrastructure. More importantly, in my opinion, it makes configuration management tools a lot more discoverable for companies that do not have dedicated system administrators. A lot of people are put off implementing configuration management because the return on investment is not always clear in advance. OpsWorks will hopefully lead to a lot more people using professional system administration practices when setting up AWS applications.

Another advantage of OpsWorks is that it further commoditises many parts of designing and deploying an application infrastructure. It is possible to find shared Chef recipes for installing common software packages such as PostgreSQL or HAProxy. Instead of manually designing your own database setup and writing custom chef recipes, you can just take a working example and tweak it if necessary.

Over time, I believe AWS will build on this platform to offer entire "off-the-shelf" application stacks. For example, a "social media stack" which contains all of the elements required to run a typical social media website, such as web servers, a database cluster and caching servers.

At the time of writing OpsWorks is still in an extended beta stage and it is not yet possible to manage all AWS resource types. As with all new AWS products, support for additional resource types is being constantly expanded.

For more information about OpsWorks, see Amazon's OpsWorks page.

Choosing a Configuration Management Package

There are a plethora of tools available in the configuration management software space. I believe the amount of different options is a result of the fact that many of their users are system administrators who can code. When a tool does not quite meet the user's needs, they begin working on their own version to scratch their individual itch, as illustrated by XKCD's Figure 4-1.
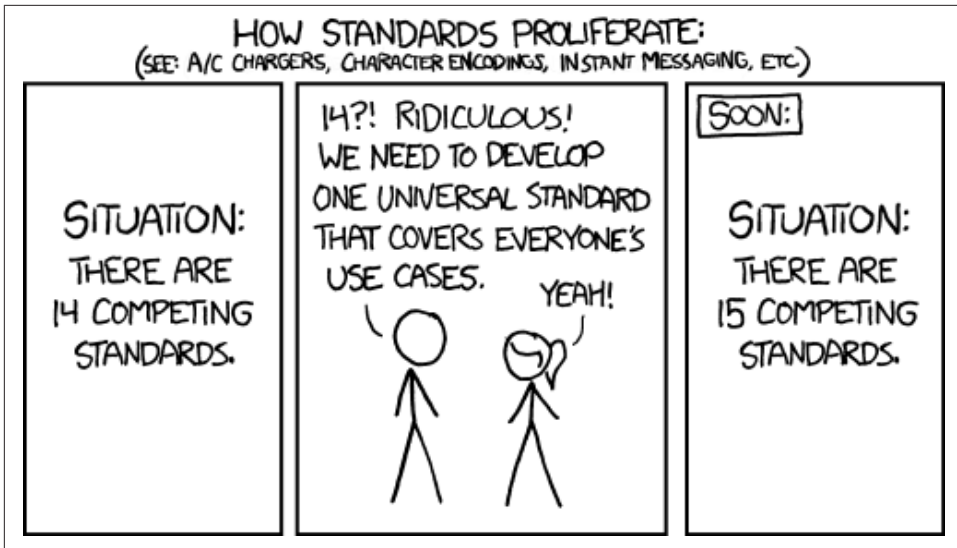
Figure 4-1. XKCD Standards

The top-tier tools all have their own approachs and architecture decisions, but share a common fundemental set of features. For example, they all provide the capability to install software packages and ensure services are running, or to create files on client hosts (such as Apache configuration files).

There are a few things to keep in mind when making a choice about which package to use.

Nearly all of the available configuration management tools have the concept of reusable pieces of code that can be shared to save even more time—instead of doing all of the work yourself, you can take some pre-written modules and customise them to fit your needs. In Puppet these are known as modules, Chef calls them recipes, and Ansible calls them playbooks.

Puppet Labs operates the Puppet Forge, a site where users can share pre-written Puppet modules for common tasks.

The availabiliy of external modules should factor in to your decision—building on the work of others will usually be easier than building from scratch.

The language used to build the tool might also come in to play, if you anticipate needing to extend it in some way (such as creating your own managed resource types). Chef and Puppet are both written in Ruby, whereas Ansible is a Python tool.

Most of the mature tools come from the pre-cloud days, and have evolved to support the needs of a more dynamic infrastructure. Chef and Puppet in particular have very good integration with AWS.

Given the relative similarity of features, choosing the right tool is in many cases merely a case of finding the one that you feel most comfortable using. Amazon's choice of Chef as the backend to OpsWorks will add to Chef's popularity in future, but that does not necessarily mean it is the right tool for everyone.

My main recommendation, especially if you are new to configuration management, is to try out a few packages and see which suits your team's workflow.

Puppet on AWS

Instead of dedicating half of the book to configuration management tools, Puppet is used to demonstrate the key concepts in the rest of this chapter. It has a good amount of overlap with other tools in terms of the available features, so all of the core principles can be applied with any of the available configuration management packages.

A Quick Introduction to Puppet

Initially launched by Luke Kanies in 2005, Puppet is one of the more popular open source configuration management tools. It uses a declarative language to let users describe the configuration and state of Unix or Windows hosts in text files known as Puppet manifests. These manifests describe the desired state of the system—*this* package should be installed, and *this* service should be running.

Typically, these manifests are stored on a central server known as the Puppet master. Client hosts periodically connect to the master server and describes their current system state. The Puppet master calculates the changes required to move from the current state to the desired state, as described in the manifests for that host, and lets the client know which changes it needs to makae. Finally, the Puppet client performs these actions.

Because clients connect to the master server at regular intervals, changes can be made on the master server and allowed to propogate throughout your network as each client connects and picks up the new configuration.

The /etc/puppet/manifests/sites.pp file is used to map server hostnames to configuration manifests, which are known as node definititions. The best way to illustrate this is with an example, which contains two node definitions:

```
# demo sites.pp with two nodes

node "www.example.com" {
    package { "nginx":
        ensure => installed
    }
}

node "db.example.com" {
    package { "postgresql":
```

```
        ensure => installed
    }
}
```

When a client named www.example.com connects to the Puppet master, the Nginx package will be installed. When db.example.com requests its configuration from the Puppet master, the PostgreSQL package will be installed.

As well as matching explicit hostnames, regular expressions can also be used in node definitions, for example "www-\d+\.example\.com" would match www-01.example.com and www-999.example.com.

Puppet supports a module-based system for creating reusable Puppet manifests. Consider the common example of needing to ensure that user accounts are automatically managed by Puppet across all of your servers. Rather than explicitly listing all of your users over and over again, they can be listed once in a "users" module which is re-used in multiple node definitions.

Modules are Puppet manifests that exist in a particular directory, usually /etc/puppet/modules. A simple "users" module, stored in /etc/puppet/modules/users/manifests/init.pp, might look like this:

```
user { "mike":
    ensure => present
}
```

This is applied to nodes by including it in their node definition, for example:

```
node "www.example.com" {
    include users
    package { "nginx":
        ensure => installed
    }
}

node "db.example.com" {
    include users
    package { "postgresql":
        ensure => installed
    }
}
```

This would ensure that a user named mike is created on both www.example.com and db.example.com.

Another way of achieving this is to use the "default" node definition, which applies to every client that connects to the Puppet master. This configuration in the default node is applied to clients which do not have an explicit node definition.

To see Puppet in action, we will launch two EC2 instances—one to act as the master, the other as a client. The initial Puppet manifest will simply install the Nginx web server package and make sure it is running—something like "Hello, World" for Puppet.

??? shows a CloudFormation stack that will create the two EC2 instances, as well as two Security Groups—these are required to access both instances with SSH, and to allow the Puppet client to contact the master on TCP port 8140.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Example Puppet master and client stack (manual install)",

    "Parameters" : {
        "KeyName": {
            "Description" : "EC2 KeyPair name",
            "Type": "String",
            "MinLength": "1",
            "MaxLength": "255",
            "AllowedPattern" : "[\\x20-\\x7E]*",
            "ConstraintDescription" : "can contain only ASCII characters."
        },
        "AMI" : {
            "Description" : "AMI ID",
            "Type": "String"
        }
    },

    "Resources": {
        "PuppetClientGroup": {
            "Type": "AWS::EC2::SecurityGroup",
            "Properties": {
                "SecurityGroupIngress": [
                    {
                        "ToPort": "22",
                        "IpProtocol": "tcp",
                        "CidrIp": "0.0.0.0/0",
                        "FromPort": "22"
                    }
                ],
                "GroupDescription": "Group for Puppet clients"
            }
        },
        "PuppetMasterGroup": {
            "Type": "AWS::EC2::SecurityGroup",
            "Properties": {
                "SecurityGroupIngress": [
                    {
                        "ToPort": "8140",
                        "IpProtocol": "tcp",
                        "SourceSecurityGroupName" : { "Ref" : "PuppetClientGroup" },
                        "FromPort": "8140"
                    },
```

```
                        {
                            "ToPort": "22",
                            "IpProtocol": "tcp",
                            "CidrIp": "0.0.0.0/0",
                            "FromPort": "22"
                        }
                    ],
                    "GroupDescription": "Group for Puppet master"
                }
            },
            "PuppetMasterInstance": {
                "Type": "AWS::EC2::Instance",
                "Properties": {
                    "ImageId" : { "Ref" : "AMI"},
                    "KeyName" : { "Ref" : "KeyName" },
                    "SecurityGroups": [
                        {
                            "Ref": "PuppetMasterGroup"
                        }
                    ],
                    "InstanceType": "t1.micro"
                }
            },
            "PuppetClientInstance": {
                "Type": "AWS::EC2::Instance",
                "Properties": {
                    "ImageId" : { "Ref" : "AMI"},
                    "KeyName" : { "Ref" : "KeyName" },
                    "SecurityGroups": [
                        {
                            "Ref": "PuppetClientGroup"
                        }
                    ],
                    "InstanceType": "t1.micro",
                    "UserData": {
                        "Fn::Base64": {
                            "Fn::GetAtt": [ "PuppetMasterInstance", "PrivateDnsName" ]
                        }
                    }
                }
            }
        },

        "Outputs" : {
            "PuppetMasterIP" : {
              "Description" : "Public IP of the Puppet master instance",
              "Value" : { "Fn::GetAtt" : [ "PuppetMasterInstance", "PublicIp" ] }
            },
            "PuppetClientIP" : {
              "Description" : "Public IP of the Puppet client instance",
              "Value" : { "Fn::GetAtt" : [ "PuppetClientInstance", "PublicIp" ] }
            }
```

```
        }
    }
```

Save this stack template to a file named *puppet-master.json*. Remember that you will need to update the KeyName and ImageId attributes. Next, create the stack with the CloudFormation command line tools, replacing the KeyName and AMI parameter values with your own:

```
aws cloudformation create-stack --region eu-west-1 --stack-name puppet-stack \
    --template-body file://puppet-stack.json \
    --parameters ParameterKey=AMI,ParameterValue=ami-4ab46b3d \
    ParameterKey=KeyName,ParameterValue=mike-ryan
```

Once the stack has been created, list the stack resources to find out the DNS names of the two instances:

```
aws cloudformation describe-stacks --stack-name puppet-test | jq '.Stacks[0].Outputs[]'
```

Now we need to install the Puppet master and create the manifests which will install Nginx on the client.

Login in to the Puppet master with SSH (using the Key Pair name you specified in the stack template), and install the Puppet master package—this will also install the Puppet client package:

```
apt-get install --yes puppet
```

With the Puppet master installed, we can begin configuration. Below is a simple example of a site.pp file, which should be saved to */etc/puppet/manifests/site.pp*.

```
node default {
    package { "nginx":
        ensure => installed
    }
    file { "/tmp/hello_world":
        ensure => present,
        content => "Hello, World!"
    }
}
```

This sites.pp file uses the default node definition, so it will be applied to any client that connects. It will install Nginx, and create a text file. Now we can move on to the client. Connect to the client instance with SSH, and install the Puppet client package:

```
apt-get install --yes puppet
```

Once installed, Puppet will run every thirty minutes by default. Unfortunately this will not work immediately—usually your Puppet master will have a more friendly DNS name such as puppet.example.com. Because we have not yet set up DNS for the Puppet master, we must use it's AWS-supplied public DNS name.

Puppet uses a key-based system for security. This provides two levels of protection—it ensures that communications between the master and clients are encrypted, and it also makes sure that only authorised clients can connect to the Puppet master and retrieve the configuration.

When a Puppet client first connects to the master, it will create a "key signing request" on the Puppet master. An administrator must authorise this request by running "*puppet sign <hostname>*", which signs the key and confirms that the client is allowed to connect and retrieve its manifests file.

On the client, run Puppet by executing the following command:

```
puppet agent --server myserver.example.com --waitforcert 120 --test
```

This commands tells Puppet that it should wait up to 120 seconds for the key to be signed on the master.

On the master, you can list the waiting requests with this command:

```
puppet cert list
```

Sign the request, updating the client's hostname to match your own:

```
puppet cert sign my-host.example.com
```

As soon as you sign the request on the master, the client should spring in to action and begin applying the configuration.

Once Puppet finishes, the client instance will have installed and started Nginx, which can be verified by checking the Nginx service is running:

```
/etc/init.d/nginx status
```

The text file will also have been created:

```
cat /tmp/testfile
```

Autoscaling and Autosign: Disabling Certificate Security

Puppet's key signing system is great when clients have a certain level of permanence, but when you are constantly creating and destroying hosts it can become an impediment. Manually signing key requests is clearly not an option when combined with AWS Autoscaling, which automatically launches instances in response to changes in required capacity.

Aware of this method of signing keys might not be suitable for all situations Puppet makes it very easy to disable with a feature known as autosigning. This is done by populating the /etc/puppet/autosign.conf file with a list of hostnames for which autosigning is enabled—when these hosts connect to the Puppet master, key checking will be bypassed. Autosign can be enabled globally by using a wildcard ("*") as the hostname.

> Disabling security measures always involves a trade-off. It is my view that, as long as your Puppet master is sufficiently protected by Security Groups or firewalls, enabling autosigning is an acceptable risk. In my experience, this is the only practical way of using Puppet in conjunction with Autoscaling, and to a lesser extent EC2 as a whole.

This example used two types of Puppet Resources—a service and a file. The Puppet Documentation site maintains a list of available Resource Types on their Type Reference page.

Puppet and CloudFormation

The previous example showed how Puppet can make it easy to manage the configuration of EC2 instances. Previous chapters has shown how CloudFormation provides a similar function for provisioning EC2 resources. What about a combination of the two?

Amazon has actually built some aspects of configuration directly in to CloudFormation —as well as creating EC2 instances, it can also automatically install packages and run services on those instances once they have launched. This means there is quite a lot of overlap between Puppet and CloudFormation, which can sometimes lead to questions over which should be responsible for particular tasks. If CloudFormation can handle many of the tasks usually handled by Puppet, do you even need to use Puppet?

CloudFormation's configuration management system works by embedding configuration data in to the stack template, via the AWS::CloudFormation::Init metadata attribute which can be specified when declaring EC2 resources in stack templates. For example, this snippet would install the puppet-server package, using the yum package manager:

```
"Resources": {
  "MyInstance": {
    "Type": "AWS::EC2::Instance",
    "Metadata" : {
      "AWS::CloudFormation::Init" : {
        "config" : {
          "packages" : {
            "yum": {
              "puppet-server": []
            }
          }
        },
[ truncated ]
```

When the instance is launched, this metadata is parsed and the required packages are installed. The metadata is parsed by the cfn-init script, which also performs the actual package installation. This script, provided by Amazon, is pre-installed on their Linux AMI and also available for installation on other operating systems.

> The cfn-init script is pre-installed on the Linux AMI, but can also be installed manually on most Linux operating systems. Amazon provides RPM packages (for RedHat-based systems, and source code for others).
>
> cfn-init is short for "CloudFormation initialisation", which should give some clues as to its purpose. It is executed during the instance's boot process, at a similar point in time to when /etc/rc.local-like scripts are executed, by passing a shell script as User Data to the instance.
>
> It is responsible performing post-boot configuration of the instance. It queries the EC2 API to find out information about the CloudFormation stack it is part of—for example, it looks at the Metadata attribute for the instance, to see which software packages should be installed.
>
> Remember that accessing the EC2 API requires the use of IAM credentials with permissions to query the APIs. Data such as tags and CloudFormation metadata is not directly available to the instance in the same way that User Data is, for example.

Because configuration information is contained within CloudFormation stack templates, it must be valid JSON. This means certain characters must be escaped to avoid invalidating the JSON, and strings can only consist of a single line (multi-line strings can be created with the Fn::Join function).

Finally, stack templates have a maximum size of 300 Kb, which acts as a hard limit to how much configuration information can be contained in a stack.

Working around these limitations means that a dedicated configuration management tool like Puppet is often easier to work with, but that does not mean CloudFormation's tools are redundant.

> Amazon provides an officially supported and maintained image, running a version of Linux based on RedHat Enterprise Linux. This image comes pre-installed with a number of AWS tools, such as the latest versions of all command line tools, and the cfn-init package.
>
> More information is available on the Linux AMI page.

Combining CloudFormation and Puppet accomplishes the majority of the aims set out in Bootstrapping an Infrastructure. CloudFormation can create the AWS resources that make up your infrastructure, and Puppet can configure the operating system and application.

Because both use text-based template files, everything can be stored in a revision control system. By checking out a copy of your repository, you have absolutely everything you need to bootstrap your infrastructure to a running state.

To demonstrate a few of CloudFormations configuration management features, and see how they interact with Puppet, we will create an example stack that automatically installs and configures a Puppet master.

This can provide the base of an entirely bootstrapped infrastructure. Information describing both resources (EC2 instances) and application-level configuration is contained within a single stack template.

We will be using the Linux AMI, which does not have Puppet "baked in"—that is, Puppet has not previously been installed when the instance launches. Instead, it will be installed by the cfn-init script which runs when the instance has finished booting.

??? shows a CloudFormation stack template which declares and EC2 instance resource, and uses its metadata to install, configure and run the Puppet master service. The template also includes some supplementary resources—the Security Groups required to access the SSH and Puppet services running on the instance.

The PuppetMasterGroup is a Security Group which will contain the Puppet master instance. It allows Puppet clients to access the master, and also allows SSH from anywhere so we can administer the instance.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Example Puppet master stack",
    "Parameters" : {
        "KeyName": {
            "Description" : "EC2 KeyPair name",
            "Type": "String",
            "MinLength": "1",
            "MaxLength": "255",
            "AllowedPattern" : "[\\x20-\\x7E]*",
            "ConstraintDescription" : "can contain only ASCII characters."
        },
        "AMI" : {
            "Description" : "AMI ID",
            "Type": "String"
        }
    },
    "Resources": {
        "CFNKeys": {❶
            "Type": "AWS::IAM::AccessKey",
            "Properties": {
                "UserName": {
                    "Ref": "CFNInitUser"
                }
            }
        }
```

```
                },
                "CFNInitUser": {
                    "Type": "AWS::IAM::User",
                    "Properties": {
                        "Policies": [
                            {
                                "PolicyName": "AccessForCFNInit",
                                "PolicyDocument": {
                                    "Statement": [
                                        {
                                            "Action": "cloudformation:DescribeStackResource",
                                            "Resource": "*",
                                            "Effect": "Allow"
                                        }
                                    ]
                                }
                            }
                        ]
                    }
                },
                "PuppetClientSecurityGroup": {❷
                    "Type": "AWS::EC2::SecurityGroup",
                    "Properties": {
                        "SecurityGroupIngress": [
                            {
                                "ToPort": "22",
                                "IpProtocol": "tcp",
                                "CidrIp": "0.0.0.0/0",
                                "FromPort": "22"
                            }
                        ],
                        "GroupDescription": "Group for clients to communicate with Puppet master"
                    }
                },
                "PuppetMasterSecurityGroup": {❸
                    "Type": "AWS::EC2::SecurityGroup",
                    "Properties": {
                        "SecurityGroupIngress": [
                            {
                                "ToPort": "8140",
                                "IpProtocol": "tcp",
                                "SourceSecurityGroupName": { "Ref": "PuppetClientSecurityGroup"},
                                "FromPort": "8140"
                            },
                            {
                                "ToPort": "22",
                                "IpProtocol": "tcp",
                                "CidrIp": "0.0.0.0/0",
                                "FromPort": "22"
                            }
                        ],
                        "GroupDescription": "Group for puppet client -> master communication"
```

```json
            }
        },
        "PuppetMasterInstance": {
            "Type": "AWS::EC2::Instance",
            "Properties": {
                "UserData": {❹
                    "Fn::Base64": {
                        "Fn::Join": [
                            "",
                            [
                                "#!/bin/bash\n",
                                "/opt/aws/bin/cfn-init --region ", { "Ref": "AWS::Region" }, " -s
                                { "Ref": "AWS::StackName" }, " -r PuppetMasterInstance ",
                                " --access-key ", { "Ref": "CFNKeys" },
                                " --secret-key ", { "Fn::GetAtt": [ "CFNKeys", "SecretAccessKey" ]
                            ]
                        ]
                    }
                },
                "KeyName": { "Ref" : "KeyName" },
                "SecurityGroups": [
                    {
                        "Ref": "PuppetMasterSecurityGroup"
                    }
                ],
                "InstanceType": "t1.micro",
                "ImageId": { "Ref" : "AMI" }❺
            },
            "Metadata": {
                "AWS::CloudFormation::Init": {
                    "config": {
                        "files": {❻
                            "/etc/puppet/autosign.conf": {
                                "content": "*.internal\n",
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/puppet/manifests/site.pp": {
                                "content": "import \"nodes\"\n",
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/puppet/manifests/nodes.pp": {
                                "content": {
                                    "Fn::Join": [
                                        "",
                                        [
                                            "node basenode {\n",
                                            " include cfn\n",
                                            "}\n",
```

```
                                         "node /^.*internal$/ inherits basenode {\n",
                                         "}\n"
                                     ]
                                 ]
                            },
                            "owner": "root",
                            "group": "wheel",
                            "mode": "100644"
                        },
                        "/etc/puppet/modules/cfn/lib/facter/cfn.rb": {
                            "owner": "root",
                            "source": "https://s3.amazonaws.com/cloudformation-examples/cfn-fa
                            "group": "wheel",
                            "mode": "100644"
                        },
                        "/etc/yum.repos.d/epel.repo": {
                            "owner": "root",
                            "source": "https://s3.amazonaws.com/cloudformation-examples/enable
                            "group": "root",
                            "mode": "000644"
                        },
                        "/etc/puppet/fileserver.conf": {
                            "content": "[modules]\n allow *.internal\n",
                            "owner": "root",
                            "group": "wheel",
                            "mode": "100644"
                        },
                        "/etc/puppet/puppet.conf": {
                            "content": {❼
                                "Fn::Join": [
                                    "",
                                    [
                                        "[main]\n",
                                        " logdir=/var/log/puppet\n",
                                        " rundir=/var/run/puppet\n",
                                        " ssldir=$vardir/ssl\n",
                                        " pluginsync=true\n",
                                        "[agent]\n",
                                        " classfile=$vardir/classes.txt\n",
                                        " localconfig=$vardir/localconfig\n"
                                    ]
                                ]
                            },
                            "owner": "root",
                            "group": "root",
                            "mode": "000644"
                        },
                        "/etc/puppet/modules/cfn/manifests/init.pp": {
                            "content": "class cfn {}",
                            "owner": "root",
                            "group": "wheel",
                            "mode": "100644"
```

```
                }
            },
            "packages": {❽
                "rubygems": {
                    "json": []
                },
                "yum": {
                    "gcc": [],
                    "rubygems": [],
                    "ruby-devel": [],
                    "make": [],
                    "puppet-server": [],
                    "puppet": []
                }
            },
            "services": {❾
                "sysvinit": {
                    "puppetmaster": {
                        "ensureRunning": "true",
                        "enabled": "true"
                    }
                }
            }
        }
    }
}
        }
    },
    "Outputs": {❿
        "PuppetMasterDNSName": {
            "Description": "Private DNS Name of PuppetMaster",
            "Value": {
                "Fn::GetAtt": [ "PuppetMasterInstance", "PrivateDnsName" ]
            }
        },
        "PuppetClientSecurityGroup": {
            "Description": "Name of the Puppet client Security Group",
            "Value": { "Ref" : "PuppetClientSecurityGroup" }
        }
    }
}
```

❶ Remember that cfn-init requires an IAM user to access the EC2 APIs. The
    CFNKeys and CFNInitUser resouces declare an IAM user with permissions to
    describe all CloudFormation stack resources, and also an IAM access key and
    secret. These credentials are passed to the Puppet master instance via User Data.
    The same result could be achieved by using IAM Roles.

❷ The PuppetClientGroup is a Security Group which will be populated with
    Puppet client instances. Any members of this group will be allowed to contact
    the Puppet master on TCP port 8140 (the default Puppet master port).

❸ The PuppetMasterGroup is a Security Group which will contain the Puppet master instance. It allows Puppet clients to access the master, and also allows SSH from anywhere so we can administer the instance.

❹ The User Data attribute for the instance is a Base64-encoded shell script. This script runs the cfn-init program, passing it some information about the stack that it is part of. This includes the EC2 Region, the stack name, and the IAM access key and secret which will be used to query the EC2 API.

Because JSON does not support multi-line strings, the Fn::Join function is used to create a mult-line shell script.

❺ You can find the latest ID for the Amazon Linux AMI at *http://aws.amazon.com/ amazon-linux-ami/*. This value is passed as a Parameter when creating the stack.

❻ Here is where the interesting part begins. Remember that the cfn-init script will retrieve this metadata once the instance has launched. The files Metadata attribute lists a number of files that will be created on the instance, along with information about the file's user and group permissions.

❻ Files can be retrieved from remote sources such as web servers or S3. They will be downloaded when the instance launches. Basic HTTP authorisation is supported.

❼ Alternatively the content can be explicitly set, using the Fn::Join function to create multi-line files.

❽ Software packages can be installed from multiple sources—this example shows some Ruby Gems being installed, along with some packages from the Yum package manager.

❾ Finally, we specify that the Puppet master service should be enabled (i.e., it should start automatically when the instance boots) and running.

❿ The Outputs section makes it easier to retrieve information about the resources in your stack. Here, we are specifying that we want to access the private DNS name of the Puppet master instance, and the name of the Puppet client Security Group.

> This example is based on Amazon's documentation—see here for further information and additional things that can be done with Puppet and CloudFormation.

Create the stack with the command line tools:

```
aws cloudformation create-stack --stack-name puppet-master --template-body file://puppet_master_cl
    --region eu-west-1 --capabilities CAPABILITY_IAM \
```

```
            --parameters ParameterKey=AMI,ParameterValue=ami-672ce210 \
            ParameterKey=KeyName,ParameterValue=mike-ryan
```

Because this stack will create an IAM user we need to add the *--capabilities CAPABIL-ITY_IAM* option to the command. Without this, CloudFormation would refuse to create the IAM User. Capabilities are used to prevent some forms of privilege escalation, as a malicious user could otherwise CloudFormation to create an IAM Policy granting them access to resources they would not otherwise have access to.

Once the stack has been created we can find out the Puppet master's DNS name and the Security Group by querying the stack's Outputs:

```
aws cloudformation describe-stacks --stack-name puppet-master | jq '.Stacks[0].Outputs[]'
```

To verify that everything is working as expected, log in to the instance with SSH and check whether the puppetmaster service is running.

Now that we know the Puppet master is up and running, we can bring up a client to demonstrate it in action.

??? shows a CloudFormation stack that declares a Puppet client instance. This instance, when launched, will connect to our Puppet master and retrieve its configuration—in this case, it will install Nginx on the client.

This stack template introduces a new feature of CloudFormation—Parameters. Parameters can be thought of as variables within your stack template. They can be given a default value which can be overriden when creating the stack. They are not quite as flexible as variables as they can only be set once (when launching or updating the stack) but they do allow for a certain amount of reusability within your stack templates.

In this example, we will use two Parameters—one for the Puppet master's DNS name, and one for the Puppet client Security Group. The Parameters to this stack are the Outputs from the previous stack.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Example Puppet client stack",
    "Parameters" : {❶
      "KeyName": {
        "Description" : "EC2 KeyPair name",
        "Type": "String",
        "MinLength": "1",
        "MaxLength": "255",
        "AllowedPattern" : "[\\x20-\\x7E]*",
        "ConstraintDescription" : "can contain only ASCII characters."
      },
      "AMI" : {
        "Description" : "AMI ID",
        "Type": "String"
      },
      "PuppetMasterDNS" : {
```

```
              "Description" : "Private DNS name of the Puppet master instance",
              "Type": "String"
          },
          "PuppetClientSecurityGroup" : {
              "Description" : "Name of the Puppet client Security Group",
              "Type": "String"
          }
      },

      "Resources": {
          "CFNKeys": {
              "Type": "AWS::IAM::AccessKey",
              "Properties": {
                  "UserName": {
                      "Ref": "CFNInitUser"
                  }
              }
          },
          "CFNInitUser": {
              "Type": "AWS::IAM::User",
              "Properties": {
                  "Policies": [
                      {
                          "PolicyName": "AccessForCFNInit",
                          "PolicyDocument": {
                              "Statement": [
                                  {
                                      "Action": "cloudformation:DescribeStackResource",
                                      "Resource": "*",
                                      "Effect": "Allow"
                                  }
                              ]
                          }
                      }
                  ]
              }
          },
          "PuppetClientInstance": {
              "Type": "AWS::EC2::Instance",
              "Properties": {
                  "UserData": {❷
                      "Fn::Base64": {
                          "Fn::Join": [
                              "",
                              [
                                  "#!/bin/bash\n",
                                  "/opt/aws/bin/cfn-init --region ", { "Ref": "AWS::Region" },  " -s
                                  { "Ref": "AWS::StackName" }, " -r PuppetClientInstance ",
                                  " --access-key ", { "Ref": "CFNKeys" },
                                  " --secret-key ", { "Fn::GetAtt": [ "CFNKeys", "SecretAccessKey" ]
                              ]
                          ]
```

```json
                    }
                },
                "KeyName": { "Ref" : "KeyName" },
                "SecurityGroups": [
                    {
                        "Ref": "PuppetClientSecurityGroup"
                    }
                ],
                "InstanceType": "t1.micro",
                "ImageId": { "Ref" : "AMI" }
            },
            "Metadata": {
                "AWS::CloudFormation::Init": {
                    "config": {
                        "files": {
                            "/etc/puppet/puppet.conf": {
                                "content": {
                                    "Fn::Join": [
                                        "",
                                        [
                                            "[main]\n",
                                            " server=", { "Ref": "PuppetMasterDNS" }, "\n",
                                            " logdir=/var/log/puppet\n",
                                            " rundir=/var/run/puppet\n",
                                            " ssldir=$vardir/ssl\n",
                                            " pluginsync=true\n",
                                            "[agent]\n",
                                            " classfile=$vardir/classes.txt\n",
                                            " localconfig=$vardir/localconfig\n"
                                        ]
                                    ]
                                },
                                "owner": "root",
                                "group": "root",
                                "mode": "000644"
                            }
                        },
                        "packages": {❸
                            "rubygems": {
                                "json": []
                            },
                            "yum": {
                                "gcc": [],
                                "rubygems": [],
                                "ruby-devel": [],
                                "make": [],
                                "puppet": []
                            }
                        },
                        "services": {❹
                            "sysvinit": {
                                "puppet": {
```

```
                              "ensureRunning": "true",
                              "enabled": "true"
                          }
                      }
                  }
              }
          }
      }
  }
}
```

❶   The Parameters define values that can be specified by the user when the stack is
      launched. If a value is not specified the default is used. If no default is set,
      attempting to create the stack without specifying a value will result in an error
      message.

❷   The User Data for this instance is the same as in the previous example: run `cfn-init`,
      `init`, and let it configure the instance according to the Metadata attributes.

❸   We only need to install the puppet package, as puppetmaster is not required on
      clients.

❹   Finally, we start the Puppet agent, which will periodically connect to the master
      and retrieve its configuration.

Because this stack uses Parameters, creating it requires a slightly different invocation of
the command line tools. We need to use the Outputs from the previous stack as Pa-
rameters, like so:

```
aws cloudformation create-stack --stack-name puppet-client --template-body file://puppet_client_cl
    --region eu-west-1 --capabilities CAPABILITY_IAM \
    --parameters ParameterKey=AMI,ParameterValue=ami-672ce210 \
    ParameterKey=KeyName,ParameterValue=mike-ryan \
    ParameterKey=PuppetMasterDNS,ParameterValue=ip-172-31-0-56.eu-west-1.compute.internal \
    ParameterKey=PuppetClientSecurityGroup,ParameterValue=puppet-master-PuppetClientSecurityGroup-
```

Once the stack has been created, we can log in to the client instance and check its status.
Remember it takes a few minutes for Puppet to run, so if you log in to the instance
immediately after it finishes booting, the Puppet run might still be in progress. Puppet
will log to */var/log/puppet.log*, so check this file if you are in any doubt.

Consider what this example achieves, in the context of Bootstrapping an Infrastructure.
We have deployed a fully-automated Puppet master, which did not require any manual
configuration whatsoever. Once that was running, we used it to bootstrap the rest of the
environment—again, without any manual configuration.

If your CloudFormation stacks are stored in (for example) GitHub, your disaster re-
covery plan can be summarised in four steps:

- Check out your GitHub repository
- Install the CloudFormation command line tools
- Launch the Puppet master stack
- Launch the Puppet client stack

This is incredibly easy to document, and the steps can be followed without requiring expertise in Puppet (or indeed, Nginx).

There are a few deficiencies in this workflow that make it somewhat inconvenient to use in production. For example, if your Puppet master's DNS name changes—which will happen if the instance is stopped and restarted, or terminated and replaced with a new instance—you will need to delete and recreate your Puppet client stack to pick up the new changes. This is clearly suboptimal—as we improve on this stack through the rest of the book, we will look at ways to make this easier to use in production. To solve the "changing DNS name" problem, we will use Route 53 to automatically assign DNS names to our instances, so our Puppet master will always be reachable at puppet.example.com.

Another potential improvement is in how files are created on the instance. The Puppet master stack demonstrates two methods—pulling files from HTTP sites, and explicitly defining the content in the stack template. There are a number of other options available, such as retrieving files from an S3 bucket.

It is worth noting that there is nothing specific to Puppet in this example—the same concept could be used to bootstrap any configuration management software, such as Chef or Ansible, or indeed any software package that can be installed with Yum, Ruby Gems, or one of the other package managers supported by CloudFormation.

There is a lot of overlap between CloudFormation and configuration management tools. While it would technically be possible to replace most of Puppet's core features with CloudFormation, it would be a lot less convenient than using a dedicated configuration management tool. If your instance configuration is very simple, CloudFormation might suffice on its own.

User Data and Tags

AWS provides to built-in mechanisms to provide data to your EC2 instances: User Data and Tags. User Data is supplied to the instance at launch time and can not be changed without restarting the instance. Tags are more flexible—these key/value pairs can be changed at any point during the instance's lifecyle.

Both of these methods can be used to provide data to your EC2 instances which can then be used by your scripts and applications. This building block enables a number of useful features. For example you can create an AMI which can perform two different

roles (e.g. running a web server or a DB server, but not both). When launching an instance from this AMI you could set a role=web or role=db tag. The launch scripts on the AMI would read this tag and know whether it should start Nginx or PostgreSQL.

Before the introduction of Tags it was necessary to build your own inventory storage system if you wanted to keep track of particular details about your instances. With Tags, EC2 is itself its own inventory system. While User Data is only available within an individual EC2 instance, Tags can be queried externally.

Tags and User Data—and the differences between them—are described in more detail in "Mapping Instances to Roles" on page 189. For now, it is worth knowing that we can use Tags both within EC2 instances, and from applications running outside of AWS, as part of an inventory management system which stores metadata for each instance.

CloudFormation also makes use of Tags. For example when an EC2 instance is created as part of a CloudFormation stack it is automatically tagged with the name and ID of the CloudFormation stack to which it belongs.

In relation to Configurmation Management tools, Tags and User Data are very useful features. Through the use of Facter plugins (which gather additional information about your systems) Puppet is able to access User Data and Tags, and use them as standard variables in its configuration manifests.

Typically, Puppet uses the hostname of the instance to decide which configuration should be applied. Because EC2 instances have auto-generated hostnames this is not immediately useful. One way to work around this problem is to use User Data to control which configuration should be applied. We will do this by providing JSON-formatted User Data which can be read by Puppet.

To begin, launch an EC2 instance with the following User Data:

```
{"role": "web"}
```

This is a JSON-formatted string simply containing a role=web key/value pair.

Once the instance is launched, install Puppet (*apt-get install puppet*), and update */etc/puppet/manifests/site.pp* with the contents shown in Example 4-1.

Example 4-1. Puppet and User Data

```
require stdlib

node default {

    $userdata = parsejson($ec2_userdata)

    $role = $userdata['role']

    case $role {
        "web": {
```

```
        require my_web_module
    }
    "db": {
        require my_database_module
    }
    default: { fail("Unrecognised role: $role") }
  }

}
```

This file is responsible for changing the behaviour of Puppet so that it ignores the host-name of the instance, and instead looks at the User Data. `node default` indiciates that the following configuration should be applied to all nodes, regardless of their hostname. We then use the `parsejson` function to read the EC2 User Data string in to a Puppet variable. Finally, we include a different module depending on the value of the `$role` variable.

You could proceed with running this example by executing puppet apply /etc/puppet/ manifests/site.pp. Because we have not yet created a my_web_module Puppet will fail. However, it will fail with a complaint that my_web_module could not be found, demonstrating the underlying theory is working as planned.

In the following section we will use Tags to look up a number of instances based on the role they are tagged with, and then execute shell scripts on the returned instances.

Executing Tasks with Fabric

The standard way to run Puppet is to allow the clients to contact the master according to a regular schedule, either using Puppet's internal scheduling (when running as a daemon) or a tool such as cron. This lets you make changes on the central Puppet server, confident in the knowledge that they will eventually propogate out to your instances.

Sometimes it can be useful to take more control over the process, and run Puppet only when you know there are changes you would like to apply. To take a common example: let us say you have a cluster of Nginx web servers behind an Elastic Load Balancer, and you would like to deploy a configuration change that will cause Nginx to restart. Allowing Puppet to run on all instances at the same time would restart all of your Nginx servers at the same time, leaving you with zero functional web servers for a brief period.

In this case it would be better to run Puppet on a few instances at a time, so that there are always enough running Nginx instances to service incoming requests.

There are cases where it is necessary to do additional work either before or after a Puppet run. Continuing with the example of web servers behind an Elastic Load Balancer—if you just need to restart Nginx it is sufficient to leave the machine "in service" (i.e. active) while Puppet runs, as Nginx will not take a long time to restart. But what if you need to perform an operation that might take a few minutes? In this scenario you will need to

remove the instance from the ELB, update it, then return it to service—not something easily achieved with Puppet alone.

There are a number of tools dedicated to simplifying the task of running commands on groups of servers. I have found Fabric to be particularly flexible when it comes to working with EC2 instances and traditional hardware alike, so I will use it for the following examples.

Fabric is a Python tool used to automate system administration tasks. It provides a basic set of operations (such as executing commands and transferring files) which can be combined with some custom logic to build powerful and flexible deployment systems, or simply make it easier to perform routine tasks on groups of servers or EC2 instances. Being Python-based, we can use Boto to quickly integrate with AWS services.

Tasks are defined by writing Python functions, which are often stored in a file named fabfile.py. These functions make use of Fabric's Python API to perform actions on remote hosts. Here is a simple example of a Fabric file supplied by fabfile.org:

```
from fabric.api import run

def host_type():
    run('uname -s')
```

The host_type task can be executed on a number of servers, for example:

```
$ fab -H localhost,linuxbox host_type
[localhost] run: uname -s
[localhost] out: Darwin
[linuxbox] run: uname -s
[linuxbox] out: Linux

Done.
Disconnecting from localhost... done.
Disconnecting from linuxbox... done.
```

Fabric understands the concept of "roles"—collections of servers, grouped by the role they perform (or some other factor). Using roles, we can easily do things like run TaskA on all web servers, followed by TaskB on all database servers.

Roles are typically defined in your Fabric file as a static collection of roles and the hostnames of their members. However they can also be created dynamically by executing Python functions which means roles can be populated by querying the AWS API with Boto. This means we can execute TaskA on all EC2 instances tagged with role=webserver, without needing to keep track of a list of instances.

To demonstrate this we will launch an instance and then use Fabric to execute commands on that host.

I have released a small Python package containing a helper function which makes it easy to look up EC2 instances using Tags. It will be used in the following example and can be downloaded from *https://github.com/mikery/fabric-ec2*.

Begin by installing Fabric and the helper library with:

```
pip install fabric
pip install git+git://github.com/mikery/fabric-ec2.git
```

Using the Web Console or Command Line Tools launch a `t1.micro` EC2 instance and provide it with some EC2 Tags. For this example I will use two Tags: "staging":"true" and "role":"web".

While the instance is launching create a file named *fabfile.py*, which will store our Fabric tasks and configuration. You could also give it another name but you will need to pass this as an option when executing Fabric, for example *fab --fabfile=/some/file.py*. The file should contain the following contents:

```
from fabric.api import run, sudo, env
from fabric_ec2 import EC2TagManager

def configure_roles(environment):
    """ Set up the Fabric env.roledefs, using the correct roles for the given environment
    """
    tags = EC2TagManager(common_tags={'staging': 'true'})

    roles = {}
    for role in ['web', 'db']:
        roles[role] = tags.get_instances(role=role)

    return roles

env.roledefs = configure_roles()

@roles('web')
def hostname():
    sudo('hostname')
```

Once the instance has launched the Fabric task can be executed with:

```
fab hostname --roles web
```

Fabric will use the EC2 API to find the hostname of any instances that match the tag query and then perform the requested task on each of them. In this case it will have only found the single test instance.

With this in place, you have a simple way of running tasks—including applying Puppet manifests—selectively across your infrastructure. Fabric provides an excellent base for automating your deployment and orchestration processes.

Because tasks are just Python functions they can be used to do a number of helpful things. Before deploying to a web server instance Fabric can first remove it from an ELB, and wait for live traffic to stop hitting the instance before it is updated.

Master-less Puppet

So far we have been working with Puppet in the typical master/client topology. This is the most common way to run Puppet, especially when working with physical hardware, outside of a cloud environment.

Puppet has another mode of operation which does not require a master server—it can operate in "local mode", where the client is responsible for applying its own configuration rather than relying on a master server.

There are two key reasons that make this useful when working with AWS.

The first is resilience. When Puppet runs during the instance boot process, it becomes a core part of your infrastructure. If the Puppet master is unavailable, autoscaling will not work properly and your application might become unavailable. Although one can deploy a cluster of Puppet masters to increase resilience, it is sometimes easier to simply remove it from the equation.

The second reason relates to autoscaling. Given that AWS can launch new instances in response to changing factors such as traffic levels, we can not predict when new instances will be launched. In environments where large numbers of instances are launched simultaneously it can be possible to overwhelm the Puppet master, leading to delays in autoscaling or even instances that fail to launch properly as Puppet is never able to complete its configuration run.

When operating in "local mode", an instance is more self-contained—it does not rely on an operational Puppet master in order to become operational itself. As a result, you have one less potential single point of failure within your infrastructure.

As always there is a trade-off to be considered—there's no such thing as a free lunch, after all. A number of useful features rely on a master/client setup, so moving to a master-less topology means these features are no longer available. These include things like Puppet Dashboard, which provides an automatically populated inventory based on data supplied by clients, and Exported Resources.

In practical terms, applying a configuration without a Puppet master is simply a case of executing:

```
puppet apply /etc/puppet/manifests/site.pp
```

This will trigger the usual Puppet logic, where the node's hostname is used to control which cconfiguration will be applied. It is also possible to apply a specific manifest:

```
puppet apply /etc/puppet/modules/mymodule/manifests/site.pp
```

This does mean that the Puppet manifests must be stored on every instance, which can be achieved in a number of ways. They can be "baked in" to the AMI if they do not change frequently, or deployed alongside your application code if they do.

We can use a modified version of ??? to create a CloudFormation stack with embedded Puppet manifests. Listing ??? shows the updated stack template.

```
{
    "AWSTemplateFormatVersion" : "2010-09-09",
    "Description": "Example Puppet local stack",
    "Parameters" : {
        "KeyName": {
            "Description" : "EC2 KeyPair name",
            "Type": "String",
            "MinLength": "1",
            "MaxLength": "255",
            "AllowedPattern" : "[\\x20-\\x7E]*",
            "ConstraintDescription" : "can contain only ASCII characters."
        },
        "AMI" : {
            "Description" : "AMI ID",
            "Type": "String"
        }
    },
    "Resources" : {
        "CFNKeys": {
            "Type": "AWS::IAM::AccessKey",
            "Properties": {
                "UserName": {
                    "Ref": "CFNInitUser"
                }
            }
        },
        "CFNInitUser": {
            "Type": "AWS::IAM::User",
            "Properties": {
                "Policies": [
                    {
                        "PolicyName": "AccessForCFNInit",
                        "PolicyDocument": {
                            "Statement": [
                                {
                                    "Action": "cloudformation:DescribeStackResource",
                                    "Resource": "*",
                                    "Effect": "Allow"
                                }
                            ]
                        }
                    }
                ]
            }
        },
```

```
"PuppetMasterInstance" : {
    "Type" : "AWS::EC2::Instance",
    "Metadata" : {
        "AWS::CloudFormation::Init" : {
            "config" : {
                "packages" : {
                    "yum" : {
                        "puppet" : [],
                        "ruby-devel" : [],
                        "gcc" : [],
                        "make" : [],
                        "rubygems" : []
                    },
                    "rubygems" : {
                        "json" : []
                    }
                },
                "files" : {
                    "/etc/yum.repos.d/epel.repo" : {
                    "source" : "https://s3.amazonaws.com/cloudformation-examples/enable-epel-c
                    "mode" : "000644",
                    "owner" : "root",
                    "group" : "root"
                    },
                    "/etc/puppet/autosign.conf" : {
                        "content" : "*.internal\n",
                        "mode" : "100644",
                        "owner" : "root",
                        "group" : "wheel"
                    },
                    "/etc/puppet/puppet.conf" : {
                        "content" : { "Fn::Join" : ["", [
                            "[main]\n",
                            " logdir=/var/log/puppet\n",
                            " rundir=/var/run/puppet\n",
                            " ssldir=$vardir/ssl\n",
                            " pluginsync=true\n",
                            "[agent]\n",
                            " classfile=$vardir/classes.txt\n",
                            " localconfig=$vardir/localconfig\n"]] },
                        "mode" : "000644",
                        "owner" : "root",
                        "group" : "root"
                    },
                    "/etc/puppet/manifests/site.pp" : {
                        "content" : { "Fn::Join" : ["", [
                            "node basenode {\n",
                            " package { 'nginx':\n",
                            "   ensure => present\n",
                            " }\n\n",
                            " service { 'nginx':\n",
                            "   ensure => running\n",
```

```
                                    "      require => Package['nginx']\n",
                                    "  }\n"]
                                    }
                                },
                                "mode" : "100644",
                                "owner" : "root",
                                "group" : "wheel"
                            }
                        }
                    }
                },
                "Properties" : {
                    "InstanceType" : "t1.micro",
                    "SecurityGroups" : [ { "Ref" : "PuppetClientGroup" } ],
                    "KeyName": { "Ref" : "KeyName" },
                    "ImageId": { "Ref" : "AMI" }
                    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
                        "#!/bin/bash\n",
                        "/opt/aws/bin/cfn-init --region ", { "Ref" : "AWS::Region" },
                        " -s ", { "Ref" : "AWS::StackName" }, " -r PuppetMasterInstance ",
                        " --access-key ", { "Ref" : "CFNKeys" },
                        " --secret-key ", { "Fn::GetAtt" : ["CFNKeys", "SecretAccessKey"]}, "\n",
                        "puppet apply /etc/puppet/manifests/site.pp", "'\n"]]}}
                }
            },
            "PuppetClientGroup" : {
                "Type" : "AWS::EC2::SecurityGroup",
                "Properties" : {
                    "SecurityGroupIngress": [
                        {
                            "ToPort": "22",
                            "IpProtocol": "tcp",
                            "CidrIp": "0.0.0.0/0",
                            "FromPort": "22"
                        }
                    ],
                    "GroupDescription" : "Security Group for Puppet clients"
                }
            }
        },
        "Outputs" : {
            "PuppetClientDNSName" : {
                "Value" : { "Fn::GetAtt" : [ "PuppetClientInstance", "PrivateDnsName" ] },
                "Description" : "DNS Name of Puppet client"
            }
        }
    }
```

The CloudFormation metadata on the NginxInstance resource ensures that the Puppet client package is installed, but that the Puppet agent service is not running. If it were, it would be regularly trying to connect to a non-existent Puppet master.

The metadata also declares the Puppet configuration files which will be created on the instance. In this example, /etc/puppet/manifests/sites.pp contains a very basic manifest that installs the Nginx web server package and ensures it is running.

Finally, the script passed as the instance's User Data runs "puppet apply" when the instance launches, applying the configuration manifest stored in sites.pp.

Creating this stack will launch an EC2 instance which automatically installs and runs Nginx, without any reliance on a Puppet master. Although the Puppet manifests used in the example were very basic, they can be expanded upon to build a more complicated infrastructure.

For more information about running Puppet in standalone mode, and other ways of scaling Puppet, see Puppet's Documentation.

Building AMIs With Packer

AMI creation is a tedious process that should be automated as soon as possible. Making AMIs manually is slow and error-prone, and installing the same packages over and over will soon get tiresome, making some type of Configuration Management tool a necessity.

This section will look at some ways to automate the process of developing and building AMIs.

When starting out with AWS, a lot of people use a very simple workflow for creating AMIs: launch an instance, manually install packages and edit configuration files, and then create an AMI (version 1).

To change the AMI the same process is followed: launch the current version of the AMI, make some configuration changes, and then create a new AMI (version 2).

This is all well and good for getting up and running quickly, but by the time you reach version 10 have a problem on your hands. Unless you have been meticulously documenting the changes you have made, it will be very difficult to repeat this process. You will have no changelog describing what was changed when, why, and by whom.

This trap should be avoided as soon as possible. You should always use a configuration management tool to help create AMIs so that the process can be easily automated. Automating this process means you can release changes faster, without wasting time manually creating AMIs.

Packer is a tool for automating the process of creating machine images. It can use a number of configuration management tools—including Chef, Puppet, and Salt—to create images for a number of platforms, including AWS.

Packer automates the process of:

- Launching a new EC2 instance

- Applying a configuration
- Creating an AMI
- Adding tags to the AMI
- Terminating the instance

Once configured, this process can be performed with a single command. Integrating Packer with continuous integration tools such as Jenkins means you can completely automate the process of creating new AMIs, and perhaps release newly-created AMIs in to your staging environment automatically.

We will use Packer to build an AMI with Nginx installed. Nginx will be installed by Puppet, demonstrating how to use configuration management tools in combination with Packer.

Begin by installing Packer according to its installation instructions.

Once Packer has been installed, create a new directory to work in, containing the sub-directories required for the Puppet manifest files:

```
mkdir packer_example
cd packer_example
mkdir -p puppet/{manifests,modules/nginx/manifests}
```

Create a file named *puppet/manifests/site.pp* with the following contents:

```
node default {
    require nginx
}
```

This will instruct Puppet to apply the Nginx class to any node using this configuration. Next create a file named *puppet/modules/nginx/manifests/init.pp* with the following contents:

```
class nginx {
    package { 'nginx':
        ensure => present
    }

    service { 'nginx':
        ensure => running
    }
}
```

This is about as simple as it gets for Puppet manifests: Nginx will be installed and configured to start when the instance boots. There are no virtual hosts configured, so Nginx will just display the default welcome page (if configured on your operating system).

For this example we will be using the Ubuntu 14.04 distribution provided by Canonical. This image does not have a recent version of Puppet pre-installed, so we must take care of that ourselves. In addition to running Puppet, Packer can also run shell scripts when

creating the image. We will use a shell script to add PuppetLab's *apt* repository and install a recent version of Puppet.

Create a file named *install_puppet.sh*, containing the text shown in Example 4-2.

Example 4-2. Puppet Install Script

```
sudo wget --quiet http://apt.puppetlabs.com/puppetlabs-release-precise.deb
sudo dpkg -i puppetlabs-release-precise.deb
sudo apt-get update
sudo apt-get remove --yes puppet puppet-common
# Install latest version of puppet from PuppetLabs repo
sudo apt-get install --yes puppet -t precise
rm puppetlabs-release-precise.deb
```

We can now move on to the Packer configuration.

It is important to understand two Packer concepts for this section:

Provisioners

These control which tools will be used to configure the image, for example Puppet or Chef. Multiple provisioners can be specified in the Packer configuration, and they will each be run sequentially.

Builders

These are the outputs of Packer. In our case, we are building an AMI. You could also build images for VMWare, OpenStack, VirtualBox, and a number of other platforms. By using multiple builders with the same provisioner configuration, it is possible to create identical machines across multiple cloud environments.

Example 4-3 shows the configuration file we will use to build our example AMI.

Example 4-3. Packer Example

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "provisioners": [
    {
      "type": "shell",
      "script": "install_puppet.sh"
    },
    { "type": "puppet-masterless",
      "manifest_file": "puppet/manifests/site.pp",
      "module_paths": ["puppet/modules"]
    }
  ],
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
```

```
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "eu-west-1",
    "source_ami": "ami-89b1a3fd",
    "instance_type": "m1.small",
    "ssh_username": "ubuntu",
    "ami_name": "my-packer-example-{{timestamp}}",
    "associate_public_ip_address": true
  }]
}
```

In the `provisioners` section we have our two provisioners: first Packer will run the shell script to install Puppet, and then it will use the `puppet-masterless` provisioner to apply the Puppet manifests.

Next we have the `builders` section, which contains a single builder object. This contains the information Packer needs to launch an EC2 instance and begin configuring it using the provisioners.

> For this example I have used Ubuntu 14.04 LTS. Applying the Puppet configuration to a vanilla installation of the operating system ensures that the instance can be recreated from scratch if necessary. It also ensures that everything required to configure the instance is contained within Puppet's configuration, as any manual changes will be removed next time the image is made.

The variables section provides your AWS credentials to Packer. If these are not hard-coded in the configuration file, Packer will attempt to read the credentials from the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables.

Save the Packer configuration file to *my_image.json*, after changing the `region` and `source_ami` parameters if desired.

To create the AMI, execute Packer with:

```
packer build my_image.json
```

Packer will be quite verbose in its output, so you can follow through as it launches the EC2 instance, runs the shell script, runs Puppet, and finally creates the AMI. The ID of the new AMI will be displayed in Packer's output.

Once Packer has finished, you can use the AWS Web Console to verify the existence of your new AMI. If you launch an instance of this image and connect to its public DNS name in your web browser, you will see the default Nginx welcome page.

This simple example demonstrates how Packer can be used to build AMIs. Even if you are not already using configuration management tools, it takes very little time to implement, and can pay for itself by reducing the amount of time you spend involved in the AMI building process.

An Example Application Stack

Because this book covers how to run a production application in AWS, it is useful to have an example application stack that can be used to demonstrate the various principles and strategies introduced in later chapters. Therefore, this chapter describes how to plan an application deployment and gradually builds up a working application using AWS components. As the chapter progresses, we will create Puppet manifests and a CloudFormation stack template that, when combined, can be used to deploy the entire application as a single self-contained stack.

By the end of this chapter you will have a web-based application running in AWS, provisioned using CloudFormation and Puppet. This application consists of multiple services: a web application, a background task processing service, a database, and a caching service.

This stack is something I have deployed many times, and is similar to those used by popular web sites such as Pinterest. You can of course replace the components I describe with your own application's components. This chapter is concerned with general processes one must go through when planning a new application, rather than the specific technologies used in this stack.

Overview of Application Components

This example deploys a Content Management System (CMS). A CMS provides a GUI-based interface for bloggers, journalists, etc. to create and update web pages without requiring knowledge of HTML or other markup languages.

The infrastructure required to operate a CMS (or indeed, most web-based applications) consists of a number of components:

Component	Role
Application Layer	Handle incoming HTTP requests.

Component	Role
Task processing layer	Perform scheduled and ad hoc application tasks.
Database layer	Provide persistent storage.
Caching layer	Provide temporary quick-access storage.

We'll build the infrastructure in this chapter using a combination of AWS services. The application and task processing layers will be handled by EC2 instances. The database layer will use AWS RDS (a hosted database service), and the caching layer will use AWS ElastiCache.

The Web Application

We will use the open-source Mezzanine CMS, which is based on Django, a Python web development framework. I chose Mezzanine because it provides a good compromise between ease of installation and customization: we get a working CMS out of the box, without having to spend too much time installing it.

The Mezzanine application will be served by the Nginx web server because it offers good performance.

Database and Caching

This application requires database and caching servers, which usually means installing software such as MySQL or Memcache. Amazon provides a number of services that act as replacements for these software packages. Instead of running your own MySQL database, you can use Amazon's Relational Database Service (RDS).

Memcache can be replaced with ElastiCache, which is a protocol-compatible replacement: that is, any valid Memcache client will work with ElastiCache without any modification.

The point of this chapter is to get an application up and running, not to spend time installing and configuring software. Therefore, the example application stack will use RDS and ElastiCache instead of installing the corresponding stand-alone software packages.

Background Task Processing

Many applications require some form of background task processing. We want our web sites to be as fast as possible from the user's perspective, so waiting around for slow tasks to complete is not an option. Today's web applications rarely live in isolation, and it is common for one web site to interact with many others through the use of external API requests.

For example, your web site might give users the opportunity to invite their friends from social networks such as Facebook or Twitter. This involves a number of API requests to query these services for a list of the users friends, and to sent out the invitations.

These API requests can take some time to complete, and making the user wait around until they do so does not make a good user experience. The best practice in this case is to move long-running tasks out of the HTTP request/response cycle and in to a dedicated background processing application. This way, the tasks can be processed asynchronously. From the user's perspective the action is completed immediately, but in fact all of the work happens in another process.

For blogs that accept comments from users, there is another popular use case for asynchronous task processing: anti-spam checks. We want to keep our blog's comments spam-free, which means every new comment must be checked by an anti-spam service.

Our example blog will use background task processing to check posted comments for spam. For this application, we will use Celery, a distributed task queue application written in Python. Celery works extremely well with Django, and is the de facto standard task processing application for many Django and Python developers.

Celery works by sending messages between your application and the processes that are actually executing your task. It requires a *message broker* to store these messages. One of the most common (and performant) Celery message brokers is RabbitMQ, which operates using the Advanced Message Queueing Protocol (AMQP).

Celery can also work with Amazon's Simple Queueing Service, which is a highly-scalable message queueing service. SQS can act as a replacement to tools like RabbitMQ. By using SQS as our Celery message broker, we do not need to install or maintain a RabbitMQ cluster.

In this infrastructure, we only have to install and maintain a single EC2 instance, which runs both the web application and the Celery task processing application. The rest of the services are provided by AWS.

Installing the Web Application

The first step is to launch and configure the EC2 instance that will run the web and task processing applications. Once it is configured and working properly, create an Amazon Machine Image so it can be used as part of a CloudFormation stack.

First we will install the software manually, and then "translate" these manual steps into Puppet manifests.

Start by creating a new Security Group named `web-instances`. This Security Group should allow inbound TCP traffic from 0.0.0.0/0 on ports 8000 and 80.

Next, launch an EC2 instance using the Ubuntu 14.04 AMI, making sure that this instance is a member of the webserver Security Group. Once the instance has launched, log in with SSH.

Mezzanine is a Python package available from PyPi, Python's package management system. Python packages can be installed with the *pip* command. First we need to install pip, along with the Python development libraries:

```
sudo apt-get install python-pip python-dev
```

Once this is done, Mezzanine itself can be installed with Pip. This will also install Django, as well as all of the other packages required to run Mezzanine.

```
sudo pip install Mezzanine
```

We need to create a directory to store the files that make up the Mezzanine project. For security purposes, we will create a new user to own this directory.

```
useradd mezzanine
sudo mkdir
sudo chown
mezzanine-project /srv/myblog
cd /srv/myblog
```

Unless otherwise configured, Mezzanine will use SQLite as its database. SQLite is a self-contained database engine: unlike MySQL or PostgreSQL, it does not require a dedicated database server. Instead, the entire database is contained within a single file, which is updated via the SQLite client libraries.

Because the database is stored in a single file, only one client can write to the database at a time. This makes it unsuitable for many production applications. But SQLite is great for development purposes, as there is no need to spend time setting up a database server when beginning the project. So we will use SQLite to make sure Mezzanine is working, before moving to an RDS-based database.

Mezzanine provides a *createdb* command that initializes the database and populates it with some example pages. The command will also prompt you to create a superuser account, which is required to access the admin interface of the site and begin making changes. Execute:

```
python manage.py createdb
```

Once this command is complete, Mezzanine is ready to run.

To ease development, Django has a built-in HTTP server that can be used to quickly test pages, without having to set up Apache or Nginx. This server can be started with:

```
python manage.py runserver
```

Open your browser, and visit the public IP address of your EC2 instance on port 8000, for example: `http://ec2-54-77-150-251.eu-west-1.compute.amazonaws.com:8000`.

You should see the Mezzanine welcome page, welcoming you to your newly-created blog.

The Django development server is not fast enough for production use, but saves plenty of time in the development phase. In production, a WSGI server such a Gunicorn is used to serve the Python application, and traffic is proxied by a web server such as Nginx or Apache. These servers are much better at dealing with higher numbers of concurrent users, while still providing fast responses to user requests.

To make this example more closely match a production environment, we will set up Nginx, and configure it to serve the Mezzanine blog application instead of using Django's development server. In this configuration, Nginx acts as a proxy to the actual application. The user's request is received by Nginx, which then forwards the request to the application server before returning the result to the client.

Nginx can communicate with the application in a few different ways, two of the most popular being HTTP and Unix sockets. Unix sockets can offer improved performance over HTTP, but they require Nginx and the application to run on the same physical server (or virtual server instance, in the case of AWS). Using HTTP to communicate with the proxy involves a little more overhead—network sockets must be created, for example—but allows Nginx and the application to run on separate servers, increasing resilience and scalability.

Install Nginx with:

```
sudo apt-get install nginx
```

Example 9 shows a simple Nginx virtual host definition. This configures Nginx to proxy traffic to an upstream application server, running on port 8000 on the same host.

Example 9. Nginx configuration

```
# nginx upstream conf, with title

upstream myblog_app {

  server localhost:8000;

}

server {
  listen              *:80 default;

  server_name         blog.example.com;
  access_log          /var/log/nginx/blog.example.com.access.log;
  location / {
    proxy_pass http://myblog_app;
    proxy_read_timeout 90;
  }
}
```

Save this configuration to */etc/nginx/sites-available/myblog.conf* and restart Nginx:

```
service nginx restart
```

If the Django development server is not already running, start it:

```
runserver ip:port
```

Now visit the public hostname of the EC2 instance in your browser again, this time on port 80. You should again see the Mezzanine welcome page, but this time it is being served by Nginx instead of Django's development server.

> If the connection times out, make sure your Security Group accepts inbound TCP traffic on port 80.

Of course, running the Django development server manually is inconvenient—we don't want to have to start it manually every time the server starts, nor do we want to have to restart it if it crashes. Therefore, stop the server and turn to our next step, which is a superserver that starts and monitors other processes.

Supervisor is a process control system that can help solve this problem. It will automatically start processes when the instance is launched, and will restart them if they crash unexpectedly. Supervisor is just one example of many tools that perform a similar function. It can be installed with:

```
sudo apt-get install supervisor
```

Example 10 shows the Supervisor configuration file required to start our Django development server. This file provides all of the information Supervisor needs to run the server. The process will be started automatically when the instance boots, and it will be automatically restarted if it crashes unexpectedly.

Example 10. Supervisor Configuration File

```
[program:myblog_app]
command=/usr/bin/python /srv/myblog/manage.py runserver
autostart=true
autorestart=unexpected
stopwaitsecs=10
user=mezzanine
```

Save this file to */etc/supervisor/conf.d/myblog_web.conf* and issue a *supervisorctl update* command, instructing Supervisor to read and process the new configuration file. Make sure you stop the manually-launched development server before doing this. Otherwise, the Supervisor-launched process will not start correctly because it will be unable to bind to port 8000, which is already in use.

```
root@web:/etc/supervisor# supervisorctl update
myblog_app: added process group
root@web:/etc/supervisor# supervisorctl status
myblog_app                      STARTING
root@web:/etc/supervisor# supervisorctl status
myblog_app                      RUNNING    pid 13012, uptime 0:00:10
root@web:/etc/supervisor#
```

Confirm that everything is working by reloading the welcome page in your web browser. Once more, the page should be displayed—only this time, the Django development server process is being managed by Supervisor. When the instance starts, the development server will be started. If the server process crashes for some reason, it will be automatically restarted.

Preparing Puppet and CloudFormation

Now that the server is configured correctly, we can retrace our steps and convert this manual process into a Puppet manifest. We will also begin creating the CloudFormation stack that describes all of the EC2 resources required to provision the application.

Puppet Files

Let's first recap the steps we took:

1. Install some packages from Apt and Pip repositories.
2. Create a directory to store the application, and a user to own the application files.
3. Initalize the application and database.
4. Create configuration files for Nginx and Supervisor.

To make this a repeatable process across any number of AWS instances, We need a Puppet module that performs all of these configuration steps. We will call this module *myblog*. The Puppet Style Guide recommends that modules consist of multiple classes, each responsible for a subset of the module's functionality. Therefore, the logic to complete the above tasks will be spread across multiple classes:

- Installing the Apt and Pip requirements will be handled by the `myblog::require ments` class.
- The logic specific to the web application server will be contained in `myblog::web`.
- Later, when we add the Celery server to the stack, its Puppet configuration will be handled by the `myblog::celery`class.

Because both the web and Celery servers have the same base set of requirements, both of these classes can include the `myblog::requirements` class instead of duplicating the requirements list.

To save time, we will use modules from the Puppet Forge where possible; this saves us from having to reinvent the wheel. Puppet modules are available for Nginx and Supervisor, and the rest of the configuration can be handled with Puppet's built-in capabilities.

> The Puppet Forge is a repository of reusable Puppet modules that you can use in your Puppet manifests. Many of these are incredibly high quality, and will give you a lot of control over how the underlying software or service is configured. You will find modules for a huge range of open source and proprietary software, as well as physical devices such as Juniper network switches.
>
> Starting in Puppet version 2.7.14, modules can be installed with the *puppet* command, for example:
>
> ```
> puppet module install puppetlabs/stdlib
> ```
>
> In previous versions, module files must be manually placed in the */etc/puppet/modules/* directory.

Begin by creating a new repository in your revision control system, and setting up the initial directory structure for Puppet's configuration files. I will be using Git for the examples.

```
git init ~/myblog
cd ~/myblog
mkdir -p puppet/{manifests,modules}
mkdir puppet/modules/myblog
```

Example 11 contains a basic *site.pp* file, used by Puppet to control which configurations are applied to each node.

Example 11.

```
require stdlib

node default {

    $userdata = parsejson($ec2_userdata)

    # Set variables from userdata
    $role = $userdata['role']

    case $role {
        "web": { $role_class = "myblog::web" }
        default: { fail("Unrecognized role: $role") }
    }

    # Main myblog class
    class { "myblog":
    }
    # Role-specific class, e.g. myblog::web
```

```
    class { $role_class:
    }

}
```

This file uses EC2 User Data, which you learned how to create in "Launching Instances" on page 12, to determine which configuration should be applied to a node. This method is described in further detail in "User Data and Tags" on page 99. Here, we are setting the $role variable from User Data. $role is then used in the case statement to set the $role_data variable to the name of the class that should be used for this node.

Finally, the main myblog class is declared, along with the role-specific class (e.g., myblog::web).

Save this file to *puppet/manifests/site.pp* and commit your changes to the repository:

```
git commit -am 'Added site.pp'
```

Next we can add the Nginx and Supervisor Puppet modules to our repository. We will do this using the git subtree command, which will pull the content of these external Git repositories in to our own repository. Do this by executing the following commands:

```
git subtree add --prefix puppet/modules/supervisor git@github.com:plathrop/puppet-module-superviso
git subtree add --prefix puppet/modules/nginx git@github.com:puppetlabs/puppetlabs-nginx.git maste
```

The Nginx module uses Puppet's stdlib (standard library) module, which provides a number of useful functions such as variable validation. This can be installed with:

```
git subtree add --prefix puppet/modules/stdlib git@github.com:puppetlabs/puppetlabs-stdlib.git mas
```

Now we can begin writing our custom module, myblog, which will make use of the Nginx and Supervisor modules, as well as performing the rest of the configuration required to run the application.

The Puppet Style Guide states that, when modules contain multiple Puppet classes, each class must be saved to its own file. This is best illustrated with an example.

Example 12 shows the top-level myblog class, which is the class we referenced in *site.pp*.

Example 12.
```
class myblog {

    $app_path = "/srv/myblog"

    class {"supervisor": }

    require myblog::requirements

}
```

This class references a subclass, `myblog::requirements`. It also sets the `$app_path` variable, which is used in the other classes. If we wanted to change the location of the Mezzanine project files, we would only need to update this variable instead of making changes in multiple files.

Save this file to *puppet/modules/myblog/manifests/init.pp*.

Example 13 contains the `myblog::requirements` class. This class installs all of the software packages required to run Mezzanine. It also creates the *mezzanine* user, and creates a directory to store the project files.

Example 13.

```
class myblog::requirements {

    $packages = ["python-dev", "python-pip"]

    package { $packages:
        ensure  => installed
    }

    $pip_packages = ["Mezzanine"]

    package { $pip_packages:
        ensure  => installed,
        provider => pip,
        require => Package[$packages]
    }

    user { "mezzanine":
        ensure  => present
    }

    file { "$myblog::app_path":
        ensure  => "directory",
        owner   => "mezzanine",
        group   => "mezzanine"
    }

}
```

Save this file to *puppet/modules/myblog/manifests/requirements.pp*.

The next subclass actually launches some servers. It is shown in Example 14.

Example 14.

```
class myblog::create_project {

    # Create the Mezzanine project
    exec { "init-mezzanine-project":
        command => "/usr/local/bin/mezzanine-project $myblog::app_path",
```

```
        user => "mezzanine",
        creates => "$myblog::app_path/__init__.py",
        notify  => Exec["init-mezzanine-db"]
    }

    # Create the development SQLite database
    exec { "init-mezzanine-db":
        command => "/usr/bin/python manage.py createdb --noinput",
        user => "mezzanine",
        cwd => "$myblog::app_path",
        refreshonly => true,
    }

}
```

This class uses Puppet's Exec resource type to execute two commands. The *mezzanine-project* command creates the initial Mezzanine project, which will set up a simple web site with a number of example pages. The *createdb* command creates the SQLite database used in development.

> Be careful when creating database resources from tools like Puppet. An incorrect configuration could result in a database being unintentionally dropped and replaced with a freshly-created database.
>
> For this reason it can be desirable to create the database outside of the configuration management tool.

The Exec resource type accepts a number of parameters that determine when it should be executed. The `init-mezzanine-project` Exec uses the `creates` parameter, which informs Puppet that executing this command will created a particular file and prevents the Exec from executing if that file already exists.

Thus, this command will execute only if */srv/myblog/__init__.py* does not exist. Because we know that `mezzanine-project` will always create this file, it is a reliable method of ensuring we do not overwrite an existing project.

The `init-mezzanine-db` Exec uses another of Puppet's control methods. It sets the `refreshonly` parameter to `true`, which means it will be executed only if explicitly requested by another resource. In this example, the `init-mezzanine-project` Exec notifies `init-mezzanine-db`, causing the latter to execute when `init-mezzanine-project` is executed.

Example 15 shows the `myblog::nginx` class:

Example 15. Nginx Puppet Module

```
class myblog::mynginx {

    class { "nginx": }
```

```
nginx::resource::upstream { "myblog_app":
    ensure  => present,
    members => [
        'localhost:8000',
    ]
}

nginx::resource::vhost { "blog.example.com":
    ensure  => enable,
    listen_options => "default",
    proxy    => "http://myblog_app",
}

}
```

Similar to the `myblog::supervisor` class, this class installs the Nginx package and writes the configuration file describing our desired Nginx setup. In this case, a single Nginx virtual host is created. This virtual host will proxy all traffic to the myblog_app proxy, which is running on port 8000.

Because we already have a class named `nginx` in the Nginx module, we can't call our class `myblog::nginx`. Instead, we call it `myblog::mynginx` to prevent a naming collision.

Example 15 should be saved to *puppet/modules/myblog/manifests/mynginx.pp*.

The final piece of the puzzle is the myblog::web class, shown in Example 16.

Example 16. myblog::web Class

```
class myblog::web {
    Class["myblog::web"] -> Class["myblog"]

    require myblog::mynginx

    supervisor::service { "myblog_web":
        ensure  => present,
        enable  => true,
        command => "/usr/bin/python ${myblog::app_path}/manage.py runserver",
        user    => "mezzanine",
        group   => "mezzanine"
    }

}
```

This class contains everything specifically related to running an application server. It imports the `myblog::nginx` class to configure the web server. It also declares a `super visor::service` resource, which will create a configuration file at */etc/supervisor/myblog_web.ini*, causing Supervisor to start the Mezzanine application when the instance launches.

Save this file to *puppet/modules/myblog/manifests/web.pp*.

Now the first step of the Puppet configuration is complete. The `myblog` module will take a fresh Ubuntu 14.04 instance and turn it in to working Mezzanine blog, served by Nginx.

CloudFormation Files

Now we set up CloudFormation to provision the EC2 instance and Security Group. Example 17 shows the first version of the CloudFormation stack.

Example 17. Initial CloudFormation Stack

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Description" : "Mezzanine-powered blog, served with Nginx.",
  "Parameters" : {
    "KeyName" : {
      "Description" : "Name of and existing EC2 KeyPair to enable SSH access to the instance",
      "Type" : "String"
    }
  },
  "Resources" : {
    "WebInstance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ],
        "KeyName" : "my-ssh-keypair",
        "ImageId" : "ami-12345678",
        "UserData": {
            "Fn::Base64": {
                "{\"role\" : \"web\"}"
            }
        }
      }
    },
    "WebSecurityGroup" : {
      "Type" : "AWS::EC2::SecurityGroup",
      "Properties" : {
        "GroupDescription" : "Allow SSH and HTTP from anywhere",
        "SecurityGroupIngress" : [
          {
            "IpProtocol" : "tcp",
            "FromPort" : "22",
            "ToPort" : "22",
            "CidrIp" : "0.0.0.0/0"
          },
          {
            "IpProtocol" : "tcp",
            "FromPort" : "80",
            "ToPort" : "80",
            "CidrIp" : "0.0.0.0/0"
          }
```

```
        ]
      }
    }
  }
}
```

This stack does not deploy the Puppet manifests to the instance when it launches—we'll add that later. Save this version of the stack template to *myblog/cloudformation/ myblog.json*.

Another point to note is the use of the `UserData` parameter on the `WebInstance` resource. This provides the instance with a JSON string describing the role for the instance, which will be used in Puppet's *manifest/site.pp* file to decide which classes to apply to the node during configuration. Because the CloudFormation manifest is itself written in JSON, we must escape quotation marks in the User Data with a backslash to ensure they are not treated as part of the manifest.

> The User Data is JSON-formatted to make it easy to read in Puppet. Because CloudFormation stacks are themselves JSON-formatted, it means the User Data must be escaped in order to be valid JSON. This, admittedly, can lead to rather ugly syntax in stack templates.
>
> An example of the User Data produced by this statement is:
>
> ```
> { "db_endpoint": "myblog.cjdrj2ogmggz.eu-west-1.rds.amazonaws.com",
>   "db_user": "awsuser",
>   "db_password": "secret123"
> }
> ```

Finally, add the Puppet module and CloudFormation stack to the Git repository.

```
cd ~/myblog
git add puppet/modules/myblog cloudformation/myblog.json
git commit -m 'added myblog module'
```

Creating an RDS Database

The Mezzanine *createdb* command used in the previous step created a SQLite database. Now that the application is working, we can replace SQLite with Amazon's Relational Database Service (RDS).

This involves two steps: creating the RDS database, and reconfiguring the Mezzanine application to use the new database instead of SQLite. First we will perform these steps manually for testing purposes, before updating the Puppet manifests and CloudForm-aton stacks to reflect the updated application.

Before we can create the RDS database, we need to create a Security Group. This procedure can protect your database instances just as it protects EC2 instances. The two

Security groups perform exactly the same function: limiting access to resources, based on the source of the request.

For our purposes, we need a Security Group that permits access from our web server instances.

Create this group by visiting the EC2 Security Groups page and clicking Create Security Group. Name the new group `db-instances` and set the description to "Myblog DB access".

After the group has been created, add an ingress rule that permits access for the `web-instances` Security Group, as shown in Figure 6. This will allow any member of the `web-instances` Security Group to access the database.

Figure 6. Creating the DB Security Group

Now we can create a new RDS database through the Web Console or command line. For the sake of example we will use the Web Console here, and later on create the same database using CloudFormation. On the RDS Console Dashboard page, click Launch a DB Instance, which will open a wizard that guides you through the process of creating a database.

The first screen presents you with a list of database engines—such as PostgreSQL, MySQL and Oracle—that can be used with RDS. Click the Select button next to MySQL Community Edition.

Figure 7 shows the second screen in the process, where the initial MySQL options are set.

Figure 7. DB Instance Details

Multi-AZ Deployment allows you to deploy a master/slave database pair that spans multiple Availability Zones. This increases resilience to failure, but also increases cost, so it is not required for this test database.

The DB Instance Identifier, Master Username, and Master Password options can be set to any values you wish—but keep track of them, as they will be needed in the next step. Then click Continue to move on to the Additional Configuration screen Figure 8.

Figure 8. Additional Configuration

Enter a name for your database (e.g., "myblog"). This database will be created automatically when the DB instance launches.

In the DB Security Groups box, select the `web-instances` DB Security Group, and click Continue.

The next screen lets you configure the automatic backup settings. For the purposes of this test I disabled automatic backups. Click Continue. After reviewing your chosen settings, and click Launch DB Instance.

Once the database instance has been created, its description will be updated to include an endpoint. This is the hostname you will use to configure your MySQL clients, for example *myblog.cjdrj2ogmggz.eu-west-1.rds.amazonaws.com*.

Now that you have a database instance running, you can reconfigure Mezzanine to use this database instead of the local SQLite database.

Mezzanine settings are controlled by a Python file located at */srv/myblog/settings.py*. Because it is quite common for different environments to require different settings (for example, a local database is used during development, and a remote database is used during production), Django and Mezzanine make it easier to override individual settings.

If a file named *local_settings.py* exists, any settings it contains will override those set in the main *settings.py* file. The settings that are consistent across all environments can be stored in *settings.py*, and any custom settings in *local_settings.py*.

> There are many ways to set different configuration options, depending on the environment in which the instance is running. The Twelve-Factor App describes one such alternative method that uses a system-level environment variable to control which settings file is used.

Example 18 shows an example *local_settings.py* file that specifies the use of an RDS database. Modify this file to reflect your database endpoint, as well as the username and password settings. The latter two should be set to the values you chose for Master Username and Master Password respectively.

Example 18.

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myblog",
        "USER": "awsuser",
        "PASSWORD": "secret123",
        "HOST": "myblog.cjdrj2ogmggz.eu-west-1.rds.amazonaws.com",
        "PORT": "",
    }
}
```

After making the required changes, save this file to */srv/myblog/local_settings.py*. We no longer need the SQLite database file, so delete that with:

```
rm /srv/myblog/dev.db
```

Because we will now be using MySQL instead of SQLite, we must ensure that the Python client libraries for MySQL are installed:

```
apt-get install python-mysqldb
```

Before we can use the new database, we must create the initial table structure and example pages by running *createdb* again:

```
python /srv/myblog/manage.py createdb
```

The Mezzanine application must be restarted to pick up the changed settings file. This is done with Supervisor's *supervisorctl* command:

```
supervisorctl restart myblog_app
```

Once the process has been restarted, use your web browser to verify that the Mezzanine site is still working.

You will probably notice that the web page feels a bit slower than it did previously. This is because the database is no longer on the same machine as the web application, which introduces some delay. After updating the Puppet and CloudFormation files with the changes we have just made, we will add a caching server to alleviate some of this delay.

RDS: Updating Puppet and CloudFormation

Now that we have completed and tested our changes manually, it is time to update the Puppet manifests to reflect them.

As part of the manual process, we created a database with the Web Console and hard-coded its endpoint in to the settings file, along with the username and password. Our end goal is to have a stack that can be deployed without any manual configuration, which means the RDS database will need to be created by CloudFormation. Hard-coding connection details will not be possible, because the database is dynamically created.

How do we solve the problem of dynamically changing configuration files based on other resources in the CloudFormation stack? "User Data and Tags" on page 99 demonstrated one way of solving this problem.

CloudFormation's `Fn::GetAtt` function can access the attributes of resources in the stack template. So we can use this function to send the database's endpoint as User Data to the instance. Puppet can then access that User Data and use it to write dynamic configuration files.

Example 19 shows an updated version of the *myblog.json* stack template.

Example 19.

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Description" : "Mezzanine-powered blog with RDS, served with Nginx.",
  "Parameters" : {
    "KeyName" : {
      "Description" : "Name of and existing EC2 KeyPair to enable SSH access to the instance",
      "Type" : "String"
    }
  },
  "Parameters" : {
    "WebAMI": {
```

```json
      "Type": "String"
    },
    "CeleryAMI": {
      "Type": "String"
    },
    "KeyPair": {
      "Type": "String"
    },
    "DBUser": {
      "Type": "String"
    },
    "DBPassword": {
      "Type": "String",
      "NoEcho": "TRUE"
    }
  },
  "Resources" : {
    "BlogDB" : {
      "Type" : "AWS::RDS::DBInstance",
      "Properties" : {
        "DBSecurityGroups" : [ {"Ref" : "DBSecurityGroup"} ],
        "DBName" : "myblog",
        "AllocatedStorage" : 5,
        "DBInstanceClass" : "t1.micro",
        "Engine" : "MySQL",
        "EngineVersion" : "5.5",
        "MasterUsername" : { "Ref" : "DBUser" },
        "MasterUserPassword" : { "Ref" : "DBPassword" }
      },
      "DeletionPolicy" : "Snapshot"
    },
    "DBSecurityGroup" : {
      "Type" : "AWS::EC2::SecurityGroup",
      "Properties" : {
        "GroupDescription" : "Allow inbound MySQL access from web instances",
        "SecurityGroupIngress" : [
          {
            "IpProtocol" : "tcp",
            "FromPort" : "3306",
            "ToPort" : "3306",
            "SourceSecurityGroupName" : { "Ref" : "WebSecurityGroup" }
          }
        ]
      }
    },
    "WebInstance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ],
        "KeyName" : { "Ref" : "KeyPair",
        "ImageId" : { "Ref" : "WebAMI" },
         "UserData" : {
           "Fn::Base64" : {
```

```
      "Fn::Join" : [ "", [
        "{\"db_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",
        " \"db_user\": \"", { "Ref": "DBUser" }, "\",",
        " \"db_password\": \"", { "Ref": "DBPassword" }, "\" }"
      ] ]
    }
   }
  }
 },
 "WebSecurityGroup" : {
```

The key items in the stack are:

1. The `BlogDB` resource is the RDS database instance, using the same settings as the database we created using the Web Console.

2. The `MasterUsername` database parameter is used in two places. First it is used when creating the RDS instance, and it is also passed to the `WebInstance` resource as part of its User Data, making it accessible to scripts running on that instance.

3. The `DeleltionProperty` database parameter ensures that when the stack is deleted, your data is not lost. Before CloudFormation terminates the database, it will perform a final snapshot.

4. The `DBSecurityGroup` allows members of the `WebSecurityGroup` to access the database instance.

5. The User Data of the `WebInstance` contains the database hostname, username, and password.

Update *~/myblog/cloudformation/myblog.json* to reflect the updates. The WebSecurityGroup is unchanged, so you don't have to update that part of the file.

Now that the database name will be sent as User Data, we need to update the Puppet manifests so that the *local_settings.py* file contains the correct endpoint, username, and password settings.

Currently, the process for setting up a new Mezzanine project with Puppet is:

1. Initialize the project with *mezzanine-project*.
2. Create the database with *createdb*.

We need to insert an additional step in the middle:

- Create a *local_settings.py* file based on the given User Data.

This must be done before running *createdb*, otherwise the default Mezzanine settings will be used, and a SQLite database will be created.

Create the new file by creating a `File` resource in the Puppet manifest, and using a template to populate the contents of this file. The template uses variables that are set in the Puppet manifests by querying the User Data for the instance.

Although this is a small change, implementing it cleanly requires changing a few of the Puppet manifest files. We could access the User Data variable directly from the `my blog::create_project` class, but this goes against Puppet's best practice guidelines.

Instead, we will convert the top-level `myblog` class to a *parameterized class*, which takes the DB endpoint, username, and password as parameters. Placing variablesin a clas is the recommended way to introduce variation into Puppet templates, as it helps make modules a lot more reusable by avoiding variable scoping issues.

Example 20 shows an updated version of the `myblog` class that accepts the required parameters.

Example 20.

```
class myblog (
    $db_endpoint, $db_user, $db_password
) {

    $app_path = "/srv/myblog"

    class {"supervisor": }

    require myblog::requirements

}
```

Update *puppet/modules/myblog/manifests/init.pp* with the contents of this example. The first line is changed to include a list of parameters that must be set when declaring an instance of the `myblog` class.

The next step is to modify *site.pp* so that it retrieves the new parameters from User Data and passes them to the `myblog` class, as shown in Example 21. The file reads these parameters from the `$userdata` variable, which is a JSON object created by reading the `$ec2_userdata` string variable. `parsejson` is a function provided by Puppet's stdlib.

Example 21.

```
require stdlib

node default {

    $userdata = parsejson($ec2_userdata)

    # Set variables from userdata
    $role = $userdata['role']
    $db_endpoint = $userdata['db_endpoint']
    $db_user = $userdata['db_user']
```

```
    $db_password = $userdata['db_password']

    case $role {
        "web": { $role_class = "myblog::web" }
        default: { fail("Unrecognized role: $role") }
    }

    # Main myblog class, takes all params
    class { "myblog":
        db_endpoint => $db_endpoint,
        db_user => $db_user,
        db_password => $db_password,
    }
    # Role-specific class, e.g. myblog::web
    class { $role_class: }

}
```

Update *puppet/manifests/site.pp* with the contents of this example.

Next we need to update the `myblog::create_project` class so that it creates the *lo cal_settings.py* file. This is shown in Example 22.

Example 22.

```
class myblog::create_project {

    # Create the Mezzanine project
    exec { "init-mezzanine-project":
        command => "/usr/local/bin/mezzanine-project $myblog::app_path",
        user => "mezzanine",
        creates => "$myblog::app_path/__init__.py",
    }

    # Create the local_settings.py file
    file { "$myblog::app_path/local_settings.py":
        ensure => present,
        content => template("myblog/local_settings.py.erb"),
        owner => "mezzanine",
        group => "mezzanine",
        require => Exec["init-mezzanine-project"],
        notify  => Exec["init-mezzanine-db"]
    }

    # Create the development SQLite database
    exec { "init-mezzanine-db":
        command => "/usr/bin/python manage.py createdb --noinput",
        user => "mezzanine",
        cwd => "$myblog::app_path",
        refreshonly => true,
    }
```

```
}
```

This file should replace *puppet/modules/myblog/create_project.pp*. The main change is to add the `File` resource that creates *local_settings.py*. Its contents will be based on the template file named *local_settings.py.erb*.

Although we specify the template name as *myblog/local_settings.py.erb*, Puppet will look for the file in *puppet/modules/myblog/templates/local_settings.py.erb*.

As before, the `require` and `notify` parameters control the ordering of Puppet resources. The *local_settings.py* file must be created before *createdb* is executed.

Finally, we need to create the template file that will be used populate the *local_settings.py* file. This is shown in Example 23.

Example 23.
```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "mydb",
        "USER": "<%= @db_user %>",
        "PASSWORD": "<%= @db_password %>",
        "HOST": "<%= @db_endpoint %>",
        "PORT": "",
    }
}
```

Save this file to *puppet/modules/myblog/templates/local_settings.py.erb*. The content is almost exactly the same as the *local_settings.py* file we created manually, except that Puppet will replace the variables with information taken from User Data.

> For more information on Puppet templates, see the documentation on Using Puppet Templates.

Commit the updated files to the Git repository:

```
git add puppet/modules/myblog/templates/local_settings.py.erb
git commit -am 'added RDS database to stack'
```

With that step complete, we can move on to the caching server.

Creating an ElastiCache Node

Caching is a required part of any high-traffic web application, so it makes sense to include some caching in the example stack.

We will use Amazon's ElastiCache service instead of running our own Memcache cluster. ElastiCache is a drop-in replacement for Memcache, which means minimal work is required on the client side.

If a cache server is specified in Mezzanine's settings file, unauthenticated page views will be automatically cached—i.e. anonymous users will see cached copies of the Mezzanine home page. To enable caching—and start using it—we simply need to let Mezzanine know there is a caching server available, which means modifying the *local_settings.py* file again.

As before, we will begin by manually setting up an ElastiCache node, and then move this configuration into the Puppet and CloudFormation files.

First, visit the EC2 Security Groups page and create a new Security Group named `myblog-cache`. Grant access to members of the web-instances EC2 Security Group as shown in Figure 9.

Figure 9. Setting Cache Security Group permissions

After creating the `mybloc-cache` Security Group and updating its permissions, go to the Cache Clusters page and click Launch Cluster to open the Launch Cluster wizard, the first screen of which is shown in Figure 10.

Figure 10. Launch Cluster wizard

The key settings here are the name of the cluster and the number of nodes. I have chosen to launch a cache cluster containing a single node, which is not ideal for reliability and resilience, but is perfect for testing.

After selecting the options shown in the screenshot, click Continue to move on to the next step. This is shown in Figure 11.

Figure 11. Setting the Security Group

This screen has only one setting that is important for our purposes: the Security Group. Select the Security Group you created (`myblog-cache`) and click Continue.

After confirming your settings on the final screen, click Launch Cache Cluster.

Once the ElastiCache cluster has launched, you will see it in the Cache Clusters list. Click on the name of the Cache Cluster to go to the description page. Here you will a list of information describing the Cache Cluster, including its *configuration endpoint*.

ElastiCache Auto Discovery

The number of nodes in an ElastiCache cluster can change at any time, due either to planned events such as launching new nodes to increase the size of the cluster, or unplanned events such as a reduction in capacity caused by a crashed node. This can lead to problems when client applications try to connect to nodes that no longer exist, or never connect to new nodes because the client is not aware of the node's existence.

To help users build more resilient applications, Amazon has extended ElastiCache with a feature known as *Auto Discovery*. This allows clients to connect to a single address—the configuration endpoint—and retrieve information about all nodes in the cluster.

Using Auto Discovery requires a compatible client, because this feature is not part of the vanilla Memcache specification. Amazon has released compatible clients for PHP and Java, and plans to add clients for other languages in future.

At a technical level, a configuration endpoint address is simply a CNAME DNS record. Resolving this address will return the hostname for one of the nodes in the ElastiCache cluster.

Amazon ensures that all nodes in the cluster contain up-to-date information about their sibling nodes. Auto Discovery compatible clients use the endpoint address to connect to one of the target nodes, from which they can retrieve a list of other nodes in the cluster.

If you are not using Java or PHP, you can still make use of Auto Discovery, albeit with a bit more effort. You will need to periodically connect to the configuration endpoint to retrieve information about members of the cluster, and update your local Memcache client configuration.

In some cases the configuration endpoint is all you need. If you maintain an ElastiCache cluster containing a single node, you can add the configuration endpoint directly to your client configuration. Because you have only one node, the configuration endpoint CNAME will always resolve to the same node address.

When running clusters with two or more nodes, using the configuration endpoint directly from an incompatible client has two drawbacks. Depending on how your client caches DNS records, traffic might become unevenly distributed between your nodes,

---

leading to under-utilized cache nodes. You will also have no control over which node your data resides on, leading to an unnecessarily high cache miss rate.

Because our cache cluster contains only a single node, we can use the configuration endpoint to configure our Memcache client.

Configuring Mezzanine to use caching is simple: if a cache host is defined in the settings file, Mezzanine will use it to case unauthenticated page views. Of course, we first need to install the Python Memcache library so Mezzanine can communicate with ElastiCache. We will use the python-memcached library. Install it with:

```
pip install python-memcached
```

The next step is to add the cache configuration information to Mezzanine's settings. Append the following text to */srv/myblog/local_settings.py*, replacing the hostname with your ElastiCache cluster's configuration endpoint:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
            "LOCATION": "myblog.ldyegh.cfg.euw1.cache.amazonaws.com:11211",
    }
}
```

With the configuration file updated, restart the Mezzanine process with:

```
supervisorctl restart myblog_app
```

Visit the Mezzanine page in your web browser. Everything should look exactly the same. However, if you refresh the page you might notice it feels faster on subsequent requests. Of course, "it feels faster" is not the most scientific of tests, so we should verify that data is in fact being written to the Memcache node. We can do this by connecting to the node with Telnet and check that it contains a record of the cached page content.

Open a connection to the node with:

```
telnet myblog.ldyegh.cfg.euw1.cache.amazonaws.com 11211
```

Memcache does not contain a built-in command to list all stored keys, so we must take a slightly roundabout approach here. Within Memcache, data is stored in a series of "slabs". We can list the slabs with the *stats slabs* command:

```
$ stats slabs
STAT 11:chunk_size 944
STAT 11:chunks_per_page 1110
STAT 11:total_pages 1
STAT 11:total_chunks 1110
STAT 11:used_chunks 1
STAT 11:free_chunks 0
STAT 11:free_chunks_end 1109
```

The number in the first column after STAT is the slab ID, in this example it is 11. We can then dump the keys that belong to that slab with the *stats cachedump* command. This accepts two arguments: the slab ID and the maximum number of keys to dump. Execute this command within your telnet session:

```
$ stats cachedump 11 100
ITEM :1:933f424fbd0e130c8d56839dc1965a1e [840 b; 1366559501 s]
END
```

A single key is dumped, verifying that our visit to the web page cached its data. You can delete this key with:

```
$ delete :1:933f424fbd0e130c8d56839dc1965a1e
```

After deleting the key, refresh the Mezzanine page and use the *cachedump* command to verify that a new key has been written.

Now that ElastiCache has been tested, we can make the relevant changes to the Puppet and CloudFormation files.

ElastiCache: Updating Puppet and CloudFormation

Because we laid much of the groundwork when setting up RDS, updating Puppet and CloudFormation to use Puppet will be a lot simpler than the previous section.

We will begin by ensuring that the python-memcached library is installed when the instance is provisioned. The *puppet/modules/myblog/manifests/requirements.pp* file contains the following line:

```
$pip_packages = ["Mezzanine"]
```

Replace this with:

```
$pip_packages = ["Mezzanine", "python-memcached"]
```

Next we need to add a new parameter to the myblog Puppet module, which will be used to store the configuration endpoint of the cache cluster. Update *puppet/modules/myblog/manifests/init.pp*, changing the class signature to:

```
class myblog ( $db_endpoint, $db_user, $db_password, $cache_endpoint ) {
```

The *puppet/manifests/site.pp* file must also be updated so that this parameter is passed when the myblog class is declared. Update this file with the following content:

```
require stdlib

node default {

    $userdata = parsejson($ec2_userdata)

    # Set variables from userdata
    $role = $userdata['role']
```

```
    $db_endpoint = $userdata['db_endpoint']
    $db_user = $userdata['db_user']
    $db_password = $userdata['db_password']
    $cache_endpoint = $userdata['cache_endpoint']

    case $role {
        "web": { $role_class = "myblog::web" }
        default: { fail("Unrecognized role: $role") }
    }

    class { "myblog":
        db_endpoint => $db_endpoint,
        db_user => $db_user,
        db_password => $db_password,
        cache_endpoint => $cache_endpoint
    }
    # Role-specific class, e.g. myblog::web
    class { $role_class: }

  }
```

Finally, update *puppet/modules/myblog/templates/local_settings.py.erb* and append the cache configuration:

```
CACHES = {
    "default": {
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
            "LOCATION": "<%= @cache_endpoint %>:11211",
    }
}
```

Those are the changes required for the Puppet side of the equation. Updating the CloudFormation stack is equally straightforward. Rather than replicating the entire stack template again, I will instead just include the sections that need to be updated.

Example 24 shows the CloudFormation stack template section that declares the resources we need to use ElastiCache: the cache cluster itself, an ElastiCache Security Group, and a Security Group ingress rule that grants our web server instance (a member of the WebSecurityGroup) access to the cache cluster nodes. Insert the text into the stack template, located at *cloudformation/myblog.json*.

Example 24.

```
"Resources" : {
  "CacheCluster": {
      "Type" : "AWS::ElastiCache::CacheCluster",
      "Properties" : {
        "CacheNodeType" : "cache.m1.small",
        "CacheSecurityGroupNames" : [ "CacheSecurityGroup" ],
        "Engine" : "memcached",
        "NumCacheNodes" : "1"
    }
```

```
    },
  "CacheSecurityGroup": {
    "Type": "AWS::ElastiCache::SecurityGroup",
    "Properties": {
      "Description"   : "Allow access from Web instances"
    }
  },
  "CacheSecurityGroupIngress": {
    "Type": "AWS::ElastiCache::SecurityGroupIngress",
    "Properties": {
      "CacheSecurityGroupName"  : { "Ref" : "CacheSecurityGroup" },
      "EC2SecurityGroupName"    : { "Ref" : "WebSecurityGroup" }
    }
  },
  "BlogDB" : {
```

Because the cache cluster configuration endpoint address is passed to the web server instances as User Data, we also need to modify the EC2 instance resource. We will again use the `Fn::GetAtt` function, this time to retrieve the configuration endpoint address.

Example 25 shows the updated User Data—note the addition of the cache_endpoint attribute. Replace the `UserData` attribute of the `WebInstance` resource with the following text:

Example 25.

```
"UserData" : {
  "Fn::Base64" : {
    "Fn::Join" : [ "", [
      "{\"db_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",
      " \"db_user\": \"", { "Ref": "DBUser" }, "\",",
      " \"db_password\": \"", { "Ref": "DBPassword" }, "\",",
      " \"cache_endpoint\": \"", { "Fn::GetAtt": [ "CacheCluster", "ConfigurationEndpoint.Address" ] }
    ] ]
  }
}
```

Now, when the stack is created, the configuration endpoint will be passed to the instance as User Data and used to populate the *local_settings.py* file.

As always, finish by committing the changes to the Git repository:

```
git commit -am 'added ElastiCache cluster'
```

Installing Celery with Simple Queueing Service

The final component in the example application stack is the background task processing service, which will be implemented using Celery.

Mezzanine does not, by default, make use of Celery, so we will briefly digress into the realm of Python programming and build a simple task that will give Celery something

useful to do as part of the demonstration. In this case, "something useful" means checking user submitted comments for spam content.

Whenever a new comment is posted on the site, Mezzanine will send a signal to notify other components in the application, letting them take action on the new comment. Signals are a feature of Django, and are an implementation of the Observer software design pattern.

We will write some code that listens for this signal. When a new comment is posted, it will be checked by our extremely simple spam filter function.

When we launch the final stack containing this infrastructure, Celery and Mezzanine will be running on separate EC2 instances. However, it is a waste of time (and money) to launch another development machine to configure Celery when we could instead use the web application instance we have used in the previous steps. So for testing purposes, perform the steps in this section on the web application instance.

Celery works by passing messages between your application processes and the background worker processes. These messages describe the task that should be executed, along with any parameters required by the task. Celery needs somewhere to store these messages, so it uses a message broker for this purpose. In keeping with the strategy of using Amazon services for as many tasks as possible in this example stack, we will use their Simple Queueing Service (SQS) as a message broker. Because Celery has built-in support for SQS, it is very simple to use.

We will begin by creating the SQS queue that will hold our messages. In the Web Console, go to the SQS page and click Create New Queue. Figure 12 shows the Create New Queue screen.

Figure 12. Creating a New SQS Queue

Enter "myblog_tasks" as the queue name, and click Create Queue.

Once the queue has been created you will be returned to the Queues page, where you can see the details of the new queue. Now we need to configure the queue's access permissions so we can read from and write to it.

Select the Permissions tab and click Add a Permission, as shown in Figure 13.

Figure 13. Queue Permissions

Figure 14 shows the next screen.

Figure 14. Adding a Queue Permission

Select the options shown in the screenshot, replacing 123456789012 with your 12-digit AWS account ID, and click Add Permission. Now anyone in your AWS account has full permissions on this SQS queue. In the next chapter, we will define a more restrictive security policy using IAM.

With the queue created, we can move on to installing and configuring Celery. First, install the django-celery library with:

```
pip install django-celery
```

This package will also install the core celery library. django-celery is a convenience package, making it easy to use Celery from within Django-powered applications such as Mezzanine.

Celery tasks are simply Python functions. By convention, these are stored in a file named *celery.py*.

Example 26 shows the Python code that handles the signal received when a new comment is posted.

Example 26. The simple asynchronous spam check

```
from celery import Celery
from django.dispatch import receiver
from django.db.models.signals import post_save
from mezzanine.generic.models import ThreadedComment

app = Celery('tasks', broker='amqp://guest@localhost//')


def is_comment_spam(comment):
    # This check is just an example!
    if "spam" in comment.comment:
        return True


@app.task
def process_comment_async(comment):
    print "Processing comment"
    if is_comment_spam(comment):
        # The comment is spam, so hide it
        ThreadedComment.objects.filter(id=comment.id).update(is_public=False)


@receiver(post_save, sender=ThreadedComment)
def process_comment(sender, **kwargs):
    process_comment_async.delay(kwargs['instance'])
```

Save this code to */srv/myblog/tasks.py*. Before moving on with the configuration, let's look at what this code is doing. This requires a brief description of a useful Django feature: signals.

Signals can be thought of as hooks for your custom code to connect to. For example, every time a database object is saved, a `post_save` signal is sent out by Django. The `@receiver` function decorator informs Django that whenever a ThreadedComment object sends the post_save signal (i.e. a new comment is posted to the blog), the `process_comment` function is called.

The `process_comment` function calls `process_Comment_async.delay`. This does not execute the code immediately—instead, it posts a message to the Celery queue. This message is picked up by a Celery worker, and the code in `process_comment_async` is executed by the worker.

This means that whenever a comment is posted to the blog it will be initially displayed. After a worker picks up the job from the queue, the message will be hidden if it contains spammy content, as defined by the `is_comment_spam` function. In this trivial case, we simply check whether the string `spam` exists in the comment text. Alas, real spammers

are not so easy to catch. You might want to update this function to perform a more reliable spam check, such as submitting the comment to Akismet's spam checking service.

Because we are using the django-celery package, we can configure Celery by updating /srv/myblogs/local_settings.py. Append the following code to that file, replacing the AWS access key and secret with your own:

Example 27. Adding Celery Settings to local_settings.py

```
BROKER_URL = 'sqs://your-aws-access-key:your-aws-access-secret@'
BROKER_TRANSPORT_OPTIONS = {'region': 'eu-west-1'}
CELERY_IMPORTS = ('tasks')

INSTALLED_APPS = (
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.redirects",
    "django.contrib.sessions",
    "django.contrib.sites",
    "django.contrib.sitemaps",
    "django.contrib.staticfiles",
    "mezzanine.boot",
    "mezzanine.conf",
    "mezzanine.core",
    "mezzanine.generic",
    "mezzanine.blog",
    "mezzanine.forms",
    "mezzanine.pages",
    "mezzanine.galleries",
    "mezzanine.twitter",
    #"mezzanine.accounts",
    #"mezzanine.mobile",
    "djcelery",
)
```

> Note the "@" at the end of the BROKER_URL setting—this is required.

Because I created my queue in the eu-west-1 region, I needed to add the `BROKER_TRANSPORT_OPTIONS` setting. If you created your queue in the default region (us-east-1), you will not need this setting.

The final Celery setting (`CELERY_IMPORTS`) makes sure that Celery loads the task from your *tasks.py* file. By convention, Celery tasks are contained within a Python module named `tasks`. Django applications can contain their own tasks module that provide

tasks relevant to the application's purpose. For example, a Django application that integrates with Twitter might provide tasks to asynchronously post new tweets or scrape profile information.

Celery can auto-discover task modules that are stored within Django applications. Because your *tasks.py* file lives in the root of the web application directory, you must explicitly import it.

INSTALLED_APPS is a core Django setting that lists the applications that make up the project: Python libraries that are automatically loaded and used by Django. In order to add djcelery, which is the Django module provided by django-celery, to this list, you need to duplicate the entire list in the *local_settings.py* file. This is one of many ways to customise Django settings.

With that done, you can launch the Celery worker process, which will wait for new tasks to be written to the SQS queue. Launch the process by running:

```
python manage.py celery worker --loglevel=info
```

If everything is configured correctly you will see some output similar to the following:

```
root@ip-10-32-34-116:/srv/myblog# python manage.py celery worker --loglevel=info
 -------------- celery@celery v3.1.13 (Cipater)
---- **** -----
--- * ***  * -- Linux-3.2.0-40-generic-x86_64-with-Ubuntu-14.04-precise
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> broker:      sqs://ABCDEFG@localhost//
- ** ---------- .> app:         default:0x298b250 (djcelery.loaders.DjangoLoader)
- ** ---------- .> concurrency: 1 (processes)
- *** --- * --- .> events:      OFF (enable -E to monitor this worker)
-- ******* ----
--- ***** ----- [queues]
 -------------- .> celery:      exchange:celery(direct) binding:celery


[Tasks]
  . myblog.tasks.process_comment_async
```

The final line of this output shows that your *tasks.py* module has been loaded successfully.

With Celery running, create a new blog post in your Mezzanine site by visiting a URL of the form *http://example.com/admin/blog/blogpost/add/* (substitute your own domain name). After creating the blog post, browse to the post's page on the main site. If your blog post title was "test", for example, this will be *http://example.com/blog/test/*.

Post a comment on the blog page by filling in the comment form. After clicking the Comment button you will see some activity in the Celery console, as the task is received and executed.

Assuming your example comment was not particularly spammy, and did not contain the string spam, it will remain on the site after the Celery task has completed. Otherwise, it will be hidden shortly after being posted.

In this example application, we are choosing to display comments by default and hide them if they prove to be spam. Another option would be to hide comments by default, and display them only if they are not spam.

Celery: Updating Puppet and CloudFormation

With Celery working, we can update the Puppet and CloudFormation configurations. This will differ slightly from the changes required to add ElastiCache and RDS, because Celery and the web application will be running on separate instances. Therefore, we need to define a new role in our Puppet manifests so that the correct processes will be started on each instance.

Begin by updating the *puppet/manifests/site.pp* file, adding celery as one of the available role types:

```
case $role {
    "web": { $role_class = "myblog::web" }
    "celery": { $role_class = "myblog::celery" }
    default: { fail("Unrecognised role: $role") }
}
```

Remember that we update $role_class to dynamically include Puppet modules based on the instance's User Data, so instances with a $role of celery will use the my blog::celery module.

Example 28 shows the myblog::celery module.

Example 28. Celery Puppet Module

```
class myblog::celery {
    Class["myblog::celery"] -> Class["myblog"]

    supervisor::service { "myblog_celery":
        ensure  => present,
        enable  => true,
        command => "/usr/bin/python ${myblog::app_path}/manage.py celery",
        user    => "mezzanine",
        group   => "mezzanine"
    }

}
```

Save this module to *puppet/modules/myblog/manifests/celery.py*. This very simple module simply ensures that Supervisor starts the Celery process. All of the heavy lifting is

done in other parts of the `myblog` module—one of the many reasons for separating Puppet modules into separate manifest files.

When installing Celery, we made some changes to the *local_settings.py* file. These changes must also be made to the template used for this file, with one modification. Append the code from Example 27 to *puppet/modules/myblog/templates/local_settings.py.erb*, and then replace the BROKER_URL setting with the following:

```
BROKER_URL = 'sqs://'
```

This removes the AWS credentials from the broker URL, telling Celery to use the keys provided by the IAM role assigned to the EC2 instance.

Now we can add Celery to the CloudFormation stack. We want to make the following changes to the stack:

1. Create an SQS queue to store Celery messages.
2. Create an IAM policy that allows web and Celery instances to write to the SQS queue.
3. Create an EC2 instance to run Celery.
4. Update the ElastiCache and RDS Security Groups to permit Celery access.
5. Update the Web EC2 instance so it can use the SQS queue name as a dynamic setting.

This requires a number of changes to *cloudformation/myblog.json*. For the sake of clarity, we will gradually update this file in a series of small steps.

Begin by adding the SQS queue.

```
"CeleryQueue": {
    "Type": "AWS::SQS::Queue"
},
```

Add this text to the `Resources` section of *myblog.json*, at the same level as `WebInstance`.

Next we will create the IAM policy and role that will set up the AWS access credentials that Celery and web instances use to access the SQS queue.

```
        "MyBlogRole": {
          "Type": "AWS::IAM::Role",
          "Properties": {
            "AssumeRolePolicyDocument": {
              "Statement": [ {
                "Effect": "Allow",
                "Principal": {
                  "Service": [ "ec2.amazonaws.com" ]
                },
                "Action": [ "sts:AssumeRole" ]
              } ]
            },
```

```
                    "Path": "/"
                }
            },
            "MyBlogRolePolicies": {
                "Type": "AWS::IAM::Policy",
                "Properties": {
                    "PolicyName": "MyBlogRole",
                    "PolicyDocument": {
                        "Statement" : [ {
                            "Effect" : "Allow",
                            "Action" : [ "sqs:*" ],
                            "Resource" : "{ "Ref" : "CeleryQueue" }"
                        } ]
                    },
                    "Roles": [ { "Ref": "MyBlogRole" } ]
                }
            },
            "MyBlogInstanceProfile": {
                "Type": "AWS::IAM::InstanceProfile",
                "Properties": {
                    "Path": "/",
                    "Roles": [ { "Ref": "MyBlogRole" } ]
                }
            },
```

Again, insert this text in to *myblog.json* at the same level as WebInstance.

The final new resources are the Celery EC2 instance and associated Security Group.

```
    "CeleryInstance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "SecurityGroups" : [ { "Ref" : "CelerySecurityGroup" } ],
        "KeyName" : "my-ssh-keypair",
        "ImageId" : { "Ref" : "CeleryAMI" },
        "IamInstanceProfile": {
            "Ref": "MyBlogInstanceProfile"
        },
        "UserData" : {
          "Fn::Base64" : {
            "Fn::Join" : [ "", [
              "{\"role\": \"celery\",",
              " \"db_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",
              " \"db_user\": \"", { "Ref": "DBUser" }, "\",",
              " \"db_password\": \"", { "Ref": "DBPassword" }, "\",",
              " \"cache_endpoint\": \"", { "Fn::GetAtt": [ "CacheCluster", "ConfigurationEndpoint.Addr
            ] ]
          }
        }
      }
    },
    "CelerySecurityGroup" : {
      "Type" : "AWS::EC2::SecurityGroup",
```

```
    "Properties" : {
      "GroupDescription" : "Allow SSH from anywhere",
      "SecurityGroupIngress" : [
        {
          "IpProtocol" : "tcp",
          "FromPort" : "22",
          "ToPort" : "22",
          "CidrIp" : "0.0.0.0/0"
        }
      ]
    }
  },
  "CelerySecurityGroupIngress": {
    "Type": "AWS::ElastiCache::SecurityGroupIngress",
    "Properties": {
      "CacheSecurityGroupName"  : { "Ref" : "CacheSecurityGroup" },
      "EC2SecurityGroupName"    : { "Ref" : "CelerySecurityGroup" }
    }
  },
```

Insert this text in to the `Resources` section of *myblog.json*, at the same level as the `WebInstance` resource. The `CeleryInstance` resource also uses a reference to the AMI Input, meaning it will use the same AMI as the `WebInstance`.

That's it for the new resources. We need to make a few other changes to this file before it is complete.

The `CelerySecurityGroupIngress` resource gives the `CeleryInstance` access to the ElastiCache cluster. We also need to allow Celery to access the RDS database instance, which requires another modification to *myblog.json*.

```
  "DBSecurityGroup" : {
    "Type" : "AWS::EC2::SecurityGroup",
    "Properties" : {
      "GroupDescription" : "Allow inbound MySQL access from web instances",
      "SecurityGroupIngress" : [
        {
          "IpProtocol" : "tcp",
          "FromPort" : "3306",
          "ToPort" : "3306",
          "SourceSecurityGroupName" : { "Ref" : "WebSecurityGroup" }
        },
        {
          "IpProtocol" : "tcp",
          "FromPort" : "3306",
          "ToPort" : "3306",
          "SourceSecurityGroupName" : { "Ref" : "CelerySecurityGroup" }
        }
      ]
    },
```

Update the DBSecurityGroup resource definition as shown here, so that the CelerySe curityGroup is listed in the DBSecurityGroupIngress attribute.

The WebInstance resource also requires some changes.

```
"WebInstance" : {
  "Type" : "AWS::EC2::Instance",
  "Properties" : {
    "SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ],
    "KeyName" : "my-ssh-keypair",
    "ImageId" : { "Ref" : "WebAMI" },
    "IamInstanceProfile": {
        "Ref": "MyBlogInstanceProfile"
    },
    "UserData" : {
      "Fn::Base64" : {
        "Fn::Join" : [ "", [
          "{\"role\": \"web\",",
          " \"db_endpoint\": \"", { "Fn::GetAtt": [ "BlogDB", "Endpoint.Address" ] }, "\",",
          " \"db_user\": \"", { "Ref": "DBUser" }, "\",",
          " \"db_password\": \"", { "Ref": "DBPassword" }, "\",",
          " \"cache_endpoint\": \"", { "Fn::GetAtt": [ "CacheCluster", "ConfigurationEndpoint.Addr
        ] ]
      }
    }
  }
},
```

This code shows the updated version of the WebInstance resourc. Note the addition of the IamInstanceProfile property, which makes the EC2 instance inherit the permissions described in the IAM policy.

With that change complete, *myblog.json* now contains a full description of our stack resources.

Building the AMIs

With the Puppet configuration complete, the next step is to create the AMI we will use for the EC2 instances. We will do this using the method described in "Building AMIs With Packer" on page 108, in the previous chapter.

For demonstration purposes we will use Packer to build two AMIs: a web AMI and a celery AMI. Because these images are very similar, you could reduce the AMI management overhead by using a single AMI for each role.

Begin by creating a directory to store the Packer configuration files. This should be at the same level as the puppet directory:

```
mkdir ~myblog/packer
```

Copy Example 4-2 to *packer/install_puppet.sh*, as described in "Building AMIs With Packer" on page 108.

First we will create the configuration for the web AMI, which is shown in Example 29. Save this file to *packer/web.json*.

Example 29. Packer Configuration: Web

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "provisioners": [
    {
      "type": "shell",
      "script": "install_puppet.sh"
    },
    { "type": "puppet-masterless",
      "manifest_file": "puppet/manifests/site.pp",
      "module_paths": ["puppet/modules"]
    }
  ],
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "eu-west-1",
    "source_ami": "ami-89b1a3fd",
    "instance_type": "m1.small",
    "ssh_username": "ubuntu",
    "associate_public_ip_address": true,
    "ami_name": "myblog-web-{{timestamp}}",
    "user_data": "{\"role\": \"web\"}"
  }]
}
```

This `amazon-ebs` object contains a `user_data` parameter. This is passed to the instance which Packer uses to create the AMI. Puppet will use this User Data to control which configuration classes are applied to the instance during the provisioning step. In this case we want to build an image for the web role, so we provide a JSON string setting the `role` to `web`.

> User Data can also be stored in a separate file, rather than cluttering up your Packer configuration file. This is especially useful when your User Data contains large JSON strings. To do this, set `user_da ta_file` to the path of the file containing your User Data.

Now we can create the Packer configuration file for the celery role. The only difference is the `role` value specified as part of the `user_data`. Copy *packer/web.json* to *packer/celery.json*, changing the `user_data` and `ami_name` to read as follows:

```
"ami_name": "myblog-celery-{{timestamp}}",
"user_data": "{\"role\": \"celery\"}"
```

With the configuration files for each role created we can now build the AMIs, starting with the web role:

```
packer build web.json
```

Once Packer finishes creating the AMI, it will output the AMI ID. Make note of this, as we will need it for the next section.

Now we can create an AMI for the Celery instance with:

```
packer build celery.json
```

Again, make a note of the AMI ID output by Packer.

With the AMIs created, we can proceed with bringing up the CloudFormation stack.

Creating the Stack with CloudFormation

Now we have created the AMIs containing our application, we can launch the Cloud-Formation stack. We will do this using the *aws* command line tool.

The stack accepts a number of parameters that we can specify on the command line. Execute the following command to begin creating the stack, replacing the parameter values where necessary:

```
aws cloudformation create-stack --region eu-west-1 --stack-name myblog-stack \
    --template-body file://myblog.json \
    --parameters ParameterKey=CeleryAMI,ParameterValue=ami-XXXXXXXX \
    ParameterKey=WebAMI,ParameterValue=ami-XXXXXXXX \
    ParameterKey=DBUser,ParameterValue=myblog_user \
    ParameterKey=DBPassword,ParameterValue=s3cr4t \
    ParameterKey=KeyName,ParameterValue=mike-ryan
```

> If there are any syntax errors in the stack template, they will be high-lighted here. Common errors include unbalanced parentheses and brackets, and misplaced commas.

Use the describe-stack-status command to check on the status of the stack:

```
aws cloudformation describe-stacks --stack-name myblog-stack | jq '.Stacks[0].StackStatus'
```

While the resources are being created, the stack's status will be `CREATE_IN_PROGRESS`. Once the resources have been successfully created this will change to `CREATED`. Any other status means an error has occured, and more information will be available in the Events tab of the Web Console, or by running the describe-stack-events command:

```
aws cloudformation describe-stack-events --stack-name myblog-stack
```

Once the stack has been created, find the public DNS name of the web instance by querying the Outputs of the stack with the describe-stacks command:

```
aws cloudformation describe-stacks --stack-name myblog-stack | jq '.Stacks[0].Outputs[]'
```

Open this address in your web browser and you should be greeted with the Mezzanine welcome page.

Adding a more friendly DNS name with Route 53 is left as an exercise for the reader.

Recap

Although Mezzanine was used an example, the core concepts in this chapter are applicable to nearly all applications.

Imagine you are a web design agency using Mezzanine as your CMS. Using the information in this chapter you could set up a test environment for new clients in a few minutes, just by creating a new stack for each client.

If you follow the process shown in this chapter, incrementally building an application by first making manual changes before "committing" those to Puppet and CloudFormation, you may save lots of debugging time.

Auto Scaling and Elastic Load Balancing

What is Auto Scaling?

Most applications have peaks and troughs of user activity. Consumer web applications are a good example. A website that is popular only in the UK is likely to experience very low levels of user activity at three o'clock in the morning, London time. Business applications also exhibit the same behavior: a company's internal HR system will see high usage during business hours, and often very little traffic outside of these times.

Capacity planning refers to the process of calculating which resources will be required to ensure that application performance remains at acceptable levels. A traditional non-cloud environment needs enough capacity to satisfy peak demand, leading to wasted resources during lulls in activity. If your application requires ten servers to satisfy peak demand and only one server during quiet times, up to nine of those servers are sometimes surplus to requirements, depending on the level of activity.

Due to the amount of time it takes to bring physical hardware online, capacity planning must take future growth in to consideration, otherwise systems administrators will spend more time ordering new hardware than configuring it. Getting these growth predictions wrong presents two risks: first, if your application fails to grow as much as expected, you have wasted a lot of money on hardware. Conversely, if you fail to anticipate explosive growth, you may find the continuation of that growth restrained by the speed in which you can bring new servers online.

Auto Scaling allows the number of provisioned instances to more closely match the demands of your application, reducing wasted resources (and therefore money).

An *Auto Scaling Group* is a collection of EC2 instances containing a dynamic number of instances. As levels of activity increase, the Auto Scaling will *scale up* by launching new instances into this group. Subsequent reductions in activity will cause Auto Scaling to *scale down* and terminate instances.

The way to actually measure the "level of activity" depends on the application. In combination with the CloudWatch monitoring service, Auto Scaling can use metrics such as CPU utilization to control scaling activities.

It is possible to submit custom metrics to CloudWatch and use these to trigger Auto Scaling events (i.e., scaling up or down). In the case of a batch processing system, you might wish to launch instances based on how many items are in the queue to be processed. Once the queue is empty, the number of running instances can be reduced to zero.

This is the "elastic" in Elastic Compute Cloud.

EC2 was not always so elastic. At the time it was launched, Amazon did not yet provide the Auto Scaling service as it exists today. Although EC2 instances could be launched and terminated on demand, performing these tasks was the responsibility of the user. As such, the pioneers who made heavy use of AWS in the early days built their own systems for managing the automatic creation and deletion of instances by interacting with the EC2 APIs.

As AWS continued to grow, Amazon built this logic in to the EC2 service under the name of Auto Scaling. Unlike some other EC2 services, it cannot be managed from the Web Console, but requires the use of command-line tools, the Auto Scaling API, or CloudFormation.

The benefits of Auto Scaling are not limited to changes in capacity—using Auto Scaling enhances the resilience of your application, and is a required component of any production-grade AWS infrastructure. Remember that Availability Zones are physically separate data centers in which EC2 instances can be launched. Auto Scaling will distribute your instances equally between AZs. In the event of a failure in one AZ, Auto Scaling will increase capacity in the remaining AZs, ensuring that your application suffers minimal disruption.

The ability to automatically scale your instances comes with its own set of potential drawbacks. For example, consider how your application will handle DDoS attacks. With a limited number of phyisical servers a concentrated attack would eventually cause your site to crash. With Auto Scaling, your instances might scale up to meet the demand of the DDoS which can become expensive very quickly. For this reason you should always impose an upper limit on the number of instances that an Auto Scaling Group can spawn, unless you are very sure that a DDoS can not break the bank as well as your application.

This chapter will introduce the core Auto Scaling concepts and put them into practice by updating the example application from the previous chapter, adding resilience and scaling capabilities to the infrastructure.

Some of the topics covered in this section include:

- How to maintain a predictable level of service by automatically replacing instances that fail
- Using notifications to keep track of instances and integrate with external management systems
- Scaling up and down automatically
- Load balancing for high availability

Static Auto Scaling Groups

Although Auto Scaling at its core revolves around the idea of dynamically increasing or reducing the number of running instances, you can also create a group with a specific number of instances which does not change dynamically. I refer to this as a *static Auto Scaling Group*.

EC2 instances can be terminated without warning for a number of reasons outside of your control; this is one of the accepted downsides of operating in the cloud. A manually-launched instance—that is, an instance launched outside of an Auto Scaling Group—would need to be replaced manually once the failure is noticed. Even if you are using CloudFormation to manage the instance, some manual interaction is required to bring the application back up.

With Auto Scaling, a failed instance is automatically replaced as soon as the failure is detected by AWS. For this reason, I always use Auto Scaling for every production instance—even if there will only ever be a single instance in the group. The small amount of extra work involved in configuring Auto Scaling is well worth the knowledge that if an entire AZ fails, the instance will be replaced without any effort on my part.

We will begin by modifying the CloudFormation stack from the previous example so that the web and Celery instances are contained within their own static Auto Scaling Group. Example 6-1 shows a CloudFormation stack template fragment describing the resources required to launch a working Auto Scaling Group.

Example 6-1. Auto Scaling Groups in CloudFormation

```
"MyLaunchConfig" : {
   "Type" : "AWS::AutoScaling::LaunchConfiguration",
   "Properties" : {
      "ImageId" : "ami-abc123",
      "SecurityGroups" : [ { "Ref" : "MySecurityGroup" } ],
      "InstanceType" : "m1.small"
   }
},
"MyASGroup" : {
   "Type" : "AWS::AutoScaling::AutoScalingGroup",
   "Properties" : {
```

```
    "AvailabilityZones" : ["eu-west-1a", "eu-west-1b", "eu-west-1c"],
    "LaunchConfigurationName" : { "Ref" : "MyLaunchConfig" },
    "MinSize" : "1",
    "MaxSize" : "1",
    "DesiredCapacity" : "1"
  }
}
```

There are two components to Auto Scaling: the Auto Scaling Group, and a *Launch Configuration*. A Launch Configuration controls which parameters are used when an instance is launched, such as the instance type and User Data.

> A full description of valid properties for an Auto Scaling Group resource can be found at the CloudFormation documentation site.

The size of an Auto Scaling Group is controlled by the MinSize and MaxSize parameters, which set lower and upper bounds on the size of the group. The DesiredCapacity parameter specifies the ideal number of instances in the group. CloudFormation will consider the Auto Scaling Group to have been created successfully only when this number of instances are running.

An Auto Scaling Group must use at least one Availability Zone. The AvailabilityZones parameter lets you control which AZs are used for the group—ideally as many as possible if resilience is a concern. Entire Availability Zones can—and have—become unavailable for extended periods of time. Amazon gives you the tools to build highly-available systems, but it is up to you to make use of them.

The parameters given to the Launch Configuration are very similar to the ones used with an EC2 instance resource. Group-level attributes—such as the number of instances in the group—are assigned to the Auto Scaling Group resource.

To update our example stack to use Auto Scaling Groups we need to perform two steps, repeated for both instance types:

1. Create a Launch Configuration to replace to EC2 resource, using the same User Data and instance type.

2. Create an Auto Scaling Group resource using this Launch Configuration.

Example 6-2 shows the Auto Scaling Groups and Launch Configurations for the web and Celery instances. Update the stack template (located at *cloudformation/ myblog.json*) by removing the WebInstance and CeleryInstance resources, and adding with the text in the example to the Resources section.

Example 6-2. Auto Scaling Web and Celery Instances

```
"CeleryLaunchConfig" : {
   "Type" : "AWS::AutoScaling::LaunchConfiguration",
   "Properties" : {
      "ImageId" : { "Ref" : "CeleryAMI" },
      "SecurityGroups" : [ { "Ref" : "CelerySecurityGroup" } ]
   }
},
"CeleryGroup" : {
   "Type" : "AWS::AutoScaling::AutoScalingGroup",
   "Properties" : {
      "AvailabilityZones" : { "Fn::GetAZs" : ""},
      "LaunchConfigurationName" : { "Ref" : "CeleryLaunchConfig" },
      "MinSize" : "1",
      "MaxSize" : "2",
      "DesiredCapacity" : "1"
   }
},

"WebLaunchConfig" : {
   "Type" : "AWS::AutoScaling::LaunchConfiguration",
   "Properties" : {
      "ImageId" : { "Ref" : "WebAMI" },
      "SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ]
   }
},
"WebGroup" : {
   "Type" : "AWS::AutoScaling::AutoScalingGroup",
   "Properties" : {
      "AvailabilityZones" : { "Fn::GetAZs" : ""},
      "LaunchConfigurationName" : { "Ref" : "WebLaunchConfig" },
      "MinSize" : "1",
      "MaxSize" : "2",
      "DesiredCapacity" : "1"
   }
},
```

The instance-specific parameters have been moved from the WebInstance resource to the WebLaunchConfig resource. The new Auto Scaling Group resource will launch one of each instance type, as set by the DesiredCapacity parameter.

The next step is to update the running CloudFormation stack with the new template. Do this using the Web Console or command-line tools, and wait for the stack to reach the *UPDATE_COMPLETE* state.

Because the WebInstance and CeleryInstance resources are no longer in the stack template, these two instances will be terminated by CloudFormation. Once the Launch Config and Auto Scaling Group resources have been created, two new instances will be launched to replace them.

It is worth noting that instances launched as part of an Auto Scaling Group are not included in the Resources panel in the Web Console. Instead, you will need to use the AWS CLI tool or the Web Console to list the members of an Auto Scaling Group. Instances will also be automatically tagged with the name and ID of the CloudFormation stack to which their parent Auto Scaling Group belongs, as well as any optional user-defined tags.

> The fact that EC2 instances are not, technically speaking, part of the CloudFormation stack has some interesting implications when updating running stacks.
>
> Consider the scenario where you wish to change the parameters of a Launch Configuration that is in use by some running instances. When you update the running stack, CloudFormation will create a new Launch Configuration, update the Auto Scaling Group to reference the new Launch Configuration, and finally delete the old Launch Configuration.
>
> By default, it will make no changes to the instances that are running at the time the stack is updated. The new Launch Configuration will apply only to newly launched instances, meaning that currently running instances will still be using the old Launch Configuration. In some cases it is acceptable to let the new Launch Configuration gradually propagate as new instances are launched. In others, it is necessary to immediately replace the instances so they pick up the new configuration.
>
> An *Update Policy* can be used to automatically replace instances when their underlying Launch Configuration is changed. Instances will be terminated in batches and replaced with new instances by the Auto Scaling service.

Now that the web and Celery instances are part of Auto Scaling Groups, we can test the resilience of our application by terminating the Celery instance via the Web Console. If you browse the Mezzanine site while the Celery instance is terminated, everything will continue to function as normal, because the web application does not rely on a functioning Celery instance in order to work, due to the decoupled nature of the application. As tasks are received, they are placed in the SQS queue, where they will wait until there is a working Celery instance to process them.

When Amazon's periodic instance health checks notice that the Celery Auto Scaling Group no longer contains a working instance, a replacement will be launched. After a few minutes, the instance will become functional and process any tasks that are waiting in the SQS queue.

With any application it is important to understand the failure characteristics of each component. How will your application cope when one or more of its components fail?

In the case of Celery, the failure characteristics are very good: the application continues working almost entirely as normal from the user's perspective. Comments posted on the blog will be delayed for a while, which many users may not even notice.

A failed `WebInstance`, on the other hand, would cause the application to become entirely unavailable, because there is only one web instance in the group. Later we will look at using load balancers to distribute traffic between multiple instances.

Notifications of Scaling Activities

Another element of AWS is the Simple Notification Service (SNS). This is a push-based notification system through which an application can publish messages to *topics*. Other applications can subscribe to these topics and receive real-time notifications when new messages are available. This can be used to implement the publish/subscribe design pattern in your application.

In addition to notifying other applications when messages are published, SNS can also send notifications to email and SMS recipients, or post the message to an external HTTP web server. Auto Scaling Groups can be optionally configured to publish SNS notifications when scaling activities take place, letting you receive an email each time new instances are launched or terminated.

Example 6-3 shows an updated version of the Celery scaling group with SNS notifications enabled. The example shows the four possible types of notifications that Auto Scaling will send. You can choose to subscribe to any combination of these types. Electing to choose all four can result in a lot of email traffic if your application regularly performs scaling activities.

Example 6-3. Auto Scaling with Notifications

```
"ScalingSNSTopic" : {
    "Type" : "AWS::SNS::Topic",
    "Properties" : {
        "Subscription" : [ {
            "Endpoint" : "notifications@example.com",
            "Protocol" : "email"
        } ]
    }
}

"CeleryGroup" : {
    "Type" : "AWS::AutoScaling::AutoScalingGroup",
    "Properties" : {
        "AvailabilityZones" : { "Fn::GetAZs" : ""},
        "LaunchConfigurationName" : { "Ref" : "CeleryLaunchConfig" },
        "MinSize" : "1",
        "MaxSize" : "2",
        "DesiredCapacity" : "1",
```

```
        "NotificationConfiguration" : {
            "TopicARN" : { "Ref" : "ScalingSNSTopic" },
            "NotificationTypes" : [
                "autoscaling:EC2_INSTANCE_LAUNCH",
                "autoscaling:EC2_INSTANCE_LAUNCH_ERROR",
                "autoscaling:EC2_INSTANCE_TERMINATE",
                "autoscaling:EC2_INSTANCE_TERMINATE_ERROR"
            ]
        }
    }
},
```

The Importance of Error Notifications

I strongly recommend subscribing to the INSTANCE_LAUNCH_ERROR notification type for any important Auto Scaling Groups. This can help alert you to issues with Auto Scaling Groups before they turn into emergencies.

Once I accidentally deleted an AMI that was still referenced in a production Launch Configuration, meaning Auto Scaling was no longer able to launch new instances.

This particular application—a social media web site—has external monitoring that measures the performance of page load times. Performance started to decrease as the running instances became increasingly overloaded. At the same time, my inbox began filling up with emails from AWS, letting me know that there was a problem with the scaling group. I quickly realized this was due to deleting the wrong AMI and set about building a new AMI. Subscribing to these notifications saved valuable time in investigating the problem.

Operator error is not the only time these messages can prove useful. If AWS is experiencing problems and can not provide an instance to satisfy an Auto Scaling request you will be informed.

Update the *cloudformation/myblog.json* file, replacing the CeleryScalingGroup resource with the one just shown. Remember to replace the example email address with your own. You could also add the NotificationConfiguration section to the WebScalingGroup resource if you would like to enable notifications for both scaling groups. After saving the file, update the running stack with the new template.

If you would like to see notifications in action, terminate the Celery instance and wait for Auto Scaling to replace it. You should receive emails for both the termination and launch events, each letting you know which instance is being terminated and the reason for the change.

Scaling Policies

Static Auto Scaling Groups have their uses, but a primary reason most people use AWS is its ability to scale compute capacity up and down on demand.

There are two ways to configure Auto Scaling to automatically change the number of instances in a group: either at fixed time intervals, or on-demand based on measurements gathered by a monitoring system.

Scaling based on time is useful only when your usage patterns are highly predictable. The implementation process for this is described in detail on Amazon's Scaling Based on a Schedule page.

Dynamic scaling is the more interesting and widely-used approach. It relies on gathering metrics—such as CPU utilization or requests per second—and using this information to decide when your application needs more or less capacity.

This is done by creating *Scaling Policies* that describe the conditions under which instances should be launched or terminated. Scaling Policies must be triggered in order to perform any action.

A policy that controls when new instances are launched is known as a *scale up policy*, and one that controls when instances are terminated is a *scale down policy*.

Scaling Policies can adjust the size of the Auto Scaling Group in three ways:

* As an exact capacity. When the policy is triggered, the number of instances will be set to a specific number defined in the policy.
* As a percentage of current capacity.
* As an absolute value. When triggered, *n* new instances will be launched, where *n* is defined in the policy.

Scaling Policies are usually triggered as a result of changes is measured metrics, which we will look at in the next section.

Scaling on CloudWatch Metrics

CloudWatch is a monitoring system provided by Amazon, tightly integrated with most AWS services. It can be used to quickly set up a custom Auto Scaling configuration specific to your application's needs. Basic metrics gathered at five minute intervals are available free of charge. Metrics can be gathered at one minute intervals, at an additional cost.

Custom metrics from third-party or self-hosted monitoring systems can be published to CloudWatch, allowing you to see this data alongside AWS-specific metrics.

CloudWatch's *Alarms* feature can be used to send alerts when these metrics fall outside of the levels that you configure. For example, you could receive an email notification when the average CPU load of an instance has been above 80% for at least ten minutes.

By connecting Alarms to Scaling Policies, CloudFormation metrics can be used to control the size of an Auto Scaling Group. Instead of informing you by email that your CPU load is too high, Amazon can launch a new instance automatically.

CloudWatch can aggregate metrics for all instances in an Auto Scaling Group and use the aggregated metrics to control scaling actions. If you have an Auto Scaling Group consisting of a cluster of instances that are all processing an equal amount of work, the average CPU utilization across the entire group will probably be a good indicator as to how busy the cluster is.

Because there are so many metrics available in CloudWatch, it is worth taking some time to evaluate different scaling strategies to see which is best for your application's work load. The right choice will depend on which resources are most heavily used by your application. Sometimes it is not possible to identify a single metric that best identifies when capacity changes are required.

Take our Celery instance as an example. The task that checks a comment for spam merely contacts an external API and returns the result to the Celery broker. This is not a particularly intensive strain on the instance's CPU because most of the task execution time will be spent waiting for responses to network requests. We could increase the parallelization of Celery by running more processes and making more efficient use of the CPU, but it is likely that the instance will run out of RAM before saturating the CPU.

Unfortunately, it is not possible to measure RAM usage in CloudWatch without writing custom scripts to submit this data to the CloudWatch API.

Because we are using the SQS service as a Celery broker, we have another option: scaling based on the number of messages in the queue, rather than on instance metrics. This is interesting because we can use one AWS service (SQS) to control another (Auto Scaling Groups), even though they are not directly connected to one another.

We will use the number of messages waiting in the SQS queue to control the size of the Celery Auto Scaling Group, ensuring there are enough instances in the group to process tasks in a timely manner at all times.

We know that our tasks are usually processed very quickly and the SQS queue is usually empty, so an increase in the queue length indicates either a legitimate increase in tasks or a problem with the Celery instances. Regardless, launching new instances will solve the problem and cause the size of the queue to decrease.

The same metric can be used to terminate instances once the queue has been reduced to an acceptable length. Running too many instances is a waste of money, so we want the Auto Scaling Group to be as small as possible.

To implement these changes we need to make further additions to the stack template, as shown in Example 6-4.

Example 6-4. Auto Scaling with CloudWatch Alarms

```
"CeleryScaleUpPolicy" : {
   "Type" : "AWS::AutoScaling::ScalingPolicy",
   "Properties" : {
      "AdjustmentType" : "ChangeInCapacity",
      "AutoScalingGroupName" : { "Ref" : "CeleryGroup" },
      "Cooldown" : "1",
      "ScalingAdjustment" : "1"
   }
},

"CeleryScaleDownPolicy" : {
   "Type" : "AWS::AutoScaling::ScalingPolicy",
   "Properties" : {
      "AdjustmentType" : "ChangeInCapacity",
      "AutoScalingGroupName" : { "Ref" : "CeleryGroup" },
      "Cooldown" : "1",
      "ScalingAdjustment" : "-1"
   }
},

"CelerySQSAlarmHigh": {
   "Type": "AWS::CloudWatch::Alarm",
   "Properties": {
      "EvaluationPeriods": "1",
      "Statistic": "Sum",
      "Threshold": "100",
      "AlarmDescription": "Triggered when SQS queue length >100",
      "Period": "60",
      "AlarmActions": [ { "Ref": "CeleryScaleUpPolicy" } ],
      "Namespace": "AWS/SQS",
```

```
      "Dimensions": [ {
         "Name": "QueueName",
         "Value": { "GetAtt": ["CeleryQueue", "QueueName"] }
      } ],
      "ComparisonOperator": "GreaterThanThreshold",
      "MetricName": "ApproximateNumberOfMessagesVisible"
   }
},

"CelerySQSAlarmLow": {
   "Type": "AWS::CloudWatch::Alarm",
   "Properties": {
      "EvaluationPeriods": "1",
      "Statistic": "Sum",
      "Threshold": "20",
      "AlarmDescription": "Triggered when SQS queue length <20",
      "Period": "60",
      "AlarmActions": [ { "Ref": "CeleryScaleDownPolicy" } ],
      "Namespace": "AWS/SQS",
      "Dimensions": [ {
         "Name": "QueueName",
         "Value": { "GetAtt": ["CeleryQueue", "QueueName"] }
      } ],
      "ComparisonOperator": "LessThanThreshold",
      "MetricName": "ApproximateNumberOfMessagesVisible"
   }
},
```

Insert this template excerpt into the Resources section of the *cloudformation/myblog.json* file.

Notice that we do not need to change any aspect of the Celery Auto Scaling Group resource in order to enable dynamic scaling. Our scaling policy configuration is entirely separate from the Auto Scaling Group to which it applies. The scaling policy could even be in a separate CloudFormation stack.

We have separate policies for scaling up and down. They both use the ChangeInCapacity adjustment type to launch or terminate a set number of instances.

The CeleryScaleUpPolicy, when triggered, will launch two new Celery instances. The CeleryScaleDownPolicy will terminate one instance at a time. Why the difference? Launching two new instances at a time lets us quickly respond to changes in demand, springing in to action as the work requirements increase. As the queue drops, we want to gradually reduce the number of instances to avoid a yo-yo effect. If we reduce the capacity of the task processing infrastructure too quickly, it can cause the queue to begin rising again, which might trigger the scale up policy. At times, the Elastic Compute Cloud can be a little too elastic.

The `Cooldown` property gives us a further means of controlling the elasticity of our Auto Scaling policy. This value, specified in seconds, imposes a delay between scaling activities to make sure the size of the group is not adjusted too frequently.

`CelerySQSAlarmHigh` is a CloudWatch Alarm resource that monitors the length of the SQS queue used for Celery tasks. When there are more than one hundred messages in the queue, this alarm is activated, triggering the `CeleryScaleUpPolicy`.

Conversely, `CelerySQSAlarmLow` triggers the `CeleryScaleDownPolicy` when the queue length drops below twenty.

In practice it is unlikely that the queue length thresholds will be so low. However, these values make it much easier to test and demonstrate that Auto Scaling is working as planned.

After saving the updated file, update the running stack with the new template. Because the `DesiredCapacity` of the group is still set to one and none of the relevant CloudWatch alarms have been triggered, nothing will actually happen yet.

To demonstrate that Auto Scaling is working, stop the Celery process on the running instance and post some test comments, causing the number of queued messages to increase until Celery is scaled up.

Using the Web Console or command-line tools, find the public DNS name of the instance in the Celery group. Remember that it will be tagged with the name of the CloudFormation stack, and `role=celery`. Log in to the instance and stop Celery with the following command:

```
supervisorctl celery stop
```

Visit the Mezzanine page in your web browser and post example comments. In another tab, open the CloudWatch Alarms page and watch the status of the `CelerySQSHighAlarm`. Once enough messages have been published to the queue, it will enter the `ALARM` state and trigger the `CeleryScaleUpPolicy`, launching two new Celery instances.

Because we configured notifications for this scaling group, you will receive a few email messages as the Auto Scaling activities are performed. After a brief period you should see there are now three running Celery instances.

Notice that they are probably all running in different Availability Zones within your region. Amazon will attempt to evenly distribute an Auto Scaling Group across an EC2 Region to enhance resilience.

The two new instances will quickly process the tasks in the queue and take the queue length below the scale down threshold. Once the `CelerySQSLowAlarm` is triggered, two of the instances will be terminated.

When terminating instances, the default behavior is to terminate the instance that has the oldest Launch Configuration. If more than one instance is running the old configuration, or all instances are running the same configuration, AWS will terminate the instance that is closest to the next instance hour. This is the most cost effective method, as it maximizes the lifetime of an instance.

If instances were launched together—as is likely in an Auto Scaling Group—more than one instance will be "closest" to a full instance hour. In this case, a random instance from this subset is terminated.

This logic can be futher controlled by assigning a Termination Policy to the Auto Scaling Group, as described in the Auto Scaling documentation.

Now that you know Auto Scaling is working, you can resume the Celery process on the original instance, assuming it was not terminated when scaling down. Do this with:

```
supervisorctl celery start
```

The Celery part of the infrastructure will now grow and shrink dynamically according to the number of tasks to be processed. Tasks will be processed as quickly as possible, while ensuring we are not wasting money by running too many instances.

Elastic Load Balancing

Whether in the cloud or on your own hardware, system failures are an inevitable part of a system administrator's life. If your application is hosted on a single server, the eventual system failure will render your application unavailable until a replacement server or virtual instance can be provisioned.

One way to improve the reliability of your application is to host it on multiple instances and distribute the traffic between them. When individual instances fail, your application will continue to run smoothly as long as the remaining instances have enough capacity to shoulder the burden of the additional requests they must now serve. This is known as *load balancing*, and the server that distributes traffic is a load balancer.

We have seen how Auto Scaling can be used to help solve this problem in AWS by automatically launching new instances to replace failed ones, or dynamically increasing capacity to respond to demand. So far we have added dynamic Auto Scaling only to the Celery part of the infrastructure.

Converting the web application instance in to an Auto Scaling group involves solving a problem not present with Celery: how do we send web requests to newly-launched instances? For testing purposes we have been using the public DNS name of each individual instance so far. Once the infrastructure is in production, users will visit by

visiting it at *http://blog.example.com*. So how can we connect a public DNS name to a group of instances?

Elastic Load Balancing is Amazon's solution to this problem. An Elastic Load Balancer (ELB) is a virtual device built specifically to provide dynamic load balancing capabilities to applications hosted on AWS. An ELB effectively sits in front of a group of EC2 instances and distributes traffic between them.

Instead of pointing your *blog.example.com* DNS record toward a specific EC2 instance, you point it at the ELB using a CNAME DNS record type. All requests will be sent, via the ELB, to the instances behind the ELB. You can use the Web Console and API to manually add and remove instances from the ELB. The ELB will regularly perform health checks on these instances. Any instances that are deemed unhealthy will be automatically removed from the ELB.

Elastic Load Balancer and Auto Scaling Groups

ELBs are designed to work on conjunction with Auto Scaling Groups. When you create a scaling group, you can specify the name of an associated ELB. New instances launched in this scaling group will be automatically placed behind the ELB, at which point they will begin receiving traffic and serving requests.

We will use this feature to convert the web application component of the example stack in to an Auto Scaling Group behind an ELB. Once finished, we will be able to access our example blog via the public DNS name of the ELB.

Example 6-5 shows the updated section of the example application stack.

Example 6-5. Auto Scaling Group with Elastic Load Balancer

```
"WebELB" : {
    "Type" : "AWS::ElasticLoadBalancing::LoadBalancer",
    "Properties" : {
      "AvailabilityZones" : { "Fn::GetAZs" : "" },
      "Listeners" : [ {
        "LoadBalancerPort" : "80",
        "InstancePort" : "80",
        "Protocol" : "HTTP"
      } ],
      "HealthCheck" : {
        "Target" : { "Fn::Join" : [ "", ["HTTP:80/"]]},
        "HealthyThreshold" : "3",
        "UnhealthyThreshold" : "5",
        "Interval" : "30",
        "Timeout" : "5"
      }
    }
},
"WebGroup" : {
```

```
    "Type" : "AWS::AutoScaling::AutoScalingGroup",
    "Properties" : {
        "AvailabilityZones" : { "Fn::GetAZs" : ""},
        "LaunchConfigurationName" : { "Ref" : "WebLaunchConfig" },
        "MinSize" : "1",
        "MaxSize" : "2",
        "DesiredCapacity" : "2",
        "LoadBalancerNames" : [ { "Ref" : "WebELB" } ]
    }
},
```

Update *cloudformation/myblog.json* to replace the `WebInstance` resource with this text. This is almost exactly the same as the changes required to convert Celery in to a scaling group, with two main changes: an Elastic Load Balancer resource has been added and the Auto Scaling Group has been updated to include the `LoadBalancerNames` parameter.

The `DesiredCapacity` for the `WebGroup` has been changed to 2, which means an additional instance will be launched when the CloudFormation stack is updated.

The ELB has a health check that verifies that the application is responding to HTTP requests.

ELB Health Checks

An ELB can be configured with a custom health check that is performed on any instances in the group. This is done by making an HTTP request to the specified target at regular intervals. If the target server does not respond to this request with a 200 status code within the given timeout, the check fails. Once the number of failed checks reaches the `UnhealthyThreshold` value, the instance is considered unhealthy and removed from the ELB. Note that an instance can be replaced if it gets very slow or unreliable, even if it doesn't fail totally.

The health check will still be performed on unhealthy instances. Should they recover from their failure, they will be automatically returned behind the ELB once enough successful checks have occurred, as specified in `HealthyThreshold`.

Another option for the custom health check is to perform a simple TCP connection. If the connection can be made, the check is a success.

HTTP-based health checks are useful for more than web applications. A useful practice is to build a custom HTTP server that represents the health of your application. This can consist of a number of tests specific to your environment, which verify that every component of the instance is working as expected. If any of these tests fail, have the server return an HTTP 500 error code in response to the health check, which results in the instance being removed from the ELB.

In addition to the custom health check just described, Amazon also performs an instance-level health check automatically. This checks for problems at the virtualization layer that might not be recognized by an application-level health check.

While health checks are useful, there are some important things to keep in mind when using them.

A misconfigured health check will cause ELB to believe all of your instance are unhealthy, and unceremoniously remove them from the group.

It is imperative that your health check return a 500 error only if the problem can be solved by removing the instance from the ELB. That is the only remedial action the ELB can take, so there is no point alerting it about problems that require another solution.

Consider the example of a web application that requires a database to function properly. If the database fails, each web instance will begin returning 500 error codes for every request. Removing an individual instance of the web server from the ELB will do nothing to bring the database back to life.

If the health check page also returns 500 error codes because the database is not working, all of the instances will be removed from the ELB. At this point visitors to your web site would see the standard ELB error page, rather than the beautifully-designed error page of your application.

Carefully monitor the number of unhealthy instances for each ELB, perhaps by setting up a CloudWatch Alarm to alert you if it remains above zero for a significant length of time.

Update the running CloudFormation stack with the modified template. The old web instance, which no longer exists in the template, will be deleted and replaced with an Auto Scaling Group with a single member.

Once the stack has finished updating, use the CloudFormation Web Console or command line tools to list the stack's Resources and find out the public DNS name of the ELB. Alternatively you could look in the Elastic Load Balancers section of the Web Console.

Visit the ELB's address in your web browser, and you should once more see the Mezzanine welcome page. This page has been served by one of the instances in the scaling group. Because they are identical, it does not matter which one actually served the request.

The number of requests received per second is one of the Elastic Load Balancing metrics in CloudWatch. As such, it can be used to scale the number of web instances in a manner directly related to current levels of traffic.

Terminate one of the running web application instances with the Web Console. You can find it by searching for one of the values contained in the instance's tags, such as the scaling group name (`WebInstanceScalingGroup`).

After this instance terminates, refresh the Mezzanine page in your browser. The site will continue to function as normal even though one of the instances is no longer running.

Within a few minutes AWS will notice the terminated instance and launch a replacement for it. Once this instance has finished, the startup process it will be marked as healthy by the ELB, and traffic will once again be split between the two instances.

With this step complete, the example application is now ready to scale at both the Celery and web levels. It is resilient to failures of individual instances and entire Availability Zones, and will run an the optimal capacity to ensure happy users and a low AWS bill.

Recap

1. If you do not use Auto Scaling, you will need to manually recover any failed instances on AWS. Even if you have only a single instance in the "group", using Auto Scaling can save a lot of headaches. When an important instance crashes, do you want to scramble to launch a new one manually, or simply read the email from Amazon letting you know that a new instance is being launched?

2. Auto Scaling policies are tricky to get right the first time, and it is likely that you will need to tweak these as your application work load changes. Err on the side of caution—in most cases it is best to run with a little extra capacity, rather than having too little capacity and offering users a poor experience.

3. Auto Scaling enables some creative thinking. Do you have an internal HR system that is only used during office hours? Use scheduled Auto Scaling to automatically launch and terminate the instance so it is only running when needed.

   What about building stacks that automatically launch when needed and "self-destruct" once their work is complete? The minimum size of an Auto Scaling Group is zero. The Celery stack could be configured to launch instances when there are more than 50 messages in the queue.

   Now you're thinking with Auto Scaling Groups.

Deployment Strategies

This chapter covers some methods that can be used to deploy changes safely and reliably to a live environment, and demonstrates two of the common approaches to updating EC2 instances:

Instance-based deployment
> Each instance is updated individually, as though it were a server in a traditional datacenter.

AMI-based deployment
> A new AMI is created every time a change is released.

The remainder of this chapter investigates the pros and cons of each approach.

In the context of this chapter, "deploying" does not just refer to updating your application's code or executable files. It's a complete, end-to-end process that makes sure your running production environment is consistent and correct. So it covers various other changes you need to manage, such as updating versions of software installed on EC2 instances and making changes to CloudFormation stack templates.

The focus here is on how to orchestrate a fleet of EC2 instances and reliably manage them. I won't cover the actions taken by the deployment script, such restarting services because each application has its own unique requirements.

Instance-based Deployments

Before we look at the AWS-specific requirements of a deploy system, let's first examine the components common to nearly all such systems.

Consider the deploy flow for a typical software company. We will look at this from the perspective of a developer making a change to some application code, but the same rules

apply to a designer changing a CSS file or a sysadmin changing a parameter in a configuration file.

1. A developer writes the code to implement a new feature.

2. The changed files are incorporated in to a source control system such as Git or Subversion.

3. Depending on the programming language being used, it might be necessary to build or compile the source files to produce an executable binary.

4. The changed source files (or compiled executables) are made available to the running instances. Instances might pull the files directly from the source control system, or perhaps use a locally-hosted repository system such as Apt or Yum.

5. Once the changed files are on the instances, running services are restarted to pick up the new code or configuration files.

This is of course a very high-level overview, and some applications might require additional steps.

Some parts of the process are not affected by the choice of hosting environment. Whether you are hosting your application on your own hardware in a shared datacenter or on AWS, you will need to perform these steps.

The elasticity of the cloud does enforce some changes on to the standard deploy flow. With Auto Scaling you can never be sure how many instances will be running when you initiate the deploy process.

Consider this example: you have ten instances running v1.0 of your application and you wish to deploy an update (v1.1). You run your deploy script and update these ten instance to v1.1. As word spreads about the amazing new features contained in this release, users flock to the site to join. The increased traffic is noticed by CloudWatch, which responds by launching two new instances to handle the load. Because these instances were not yet launched when the deploy was executed they will be launched with whichever version of the application was baked in to the AMI—in this case, v1.0. As a result you end up running two versions of your application simultaneously.

To solve this problem, each instance must be able to update itself to the latest released version at launch time, and finish updating itself before it is added to the pool of instances behind an Elastic Load Balancer.

This approach is referred to as an *instance-based deploy*, and is very similar to release management processes in non-cloud environments. Two additional features are required to make it AWS-compatible: finding out the hostnames of instances which should be updated, and making sure instances can update themselves on boot.

Executing Code on Running Instances with Fabric

The first problem when deploying code to running instances is a simple one: how can we reliably execute code on running EC2 instances when we don't know their hostnames in advance?

Part of the answer to that question is Fabric, which is a Python tool used to automate system administration tasks. It provides a basic set of operations (such as executing commands and transferring files) that can be combined with some custom logic to build powerful and flexible deployment systems, or simply make it easier to perform routine tasks on groups of servers or EC2 instances.

Because Fabric is Python-based, we can use Boto to quickly integrate it with AWS services. Tasks are defined by writing Python functions, which are often stored in a file named *fabfile.py*. These functions make use of Fabric's Python API to perform actions on remote hosts. A simple example is:

```
from fabric.api import run

def get_uptime():
    run('uptime')
```

When executed, this task will run the *uptime* command on each host and display the resulting output. It can be executed in a number of ways, for example:

```
fab -H localhost,www.example.com get_uptime
```

With this invocation, Fabric would execute the `get_uptime` task—and therefore the *uptime* command—on both *localhost* and *www.example.com*. The `-H` flag defines the list of hosts on which the task will be executed.

Grouping instances through roles

Fabric includes a feature known as *roles*, which are user-defined groups of hosts. The Roles Documentation shows a simple example:

```
from fabric.api import env

env.roledefs = {
    'web': ['www1', 'www2', 'www3'],
    'dns': ['ns1', 'ns2']
}
```

A role is simply a list of hostnames (or, technically speaking, a list of "host strings," which may or may not be fully qualified domain names). As the previous code shows, the role definition list—`env.roledefs`—is implemented as a Python dictionary, where a key such as `'web'` is associated to an array of host strings (`www1`, `www2`, and `www3`). We make it available to other parts of the Python script through the global `env` variable.

When you combine the code just shown with the previous example, our own `get_up time` task could be executed on the three web servers by executing this command:

```
fab -R web get_uptime
```

This command would execute `get_uptime` on the three web servers listed under `env.ro ledefs` as `'web'` and display the output, making it functionally equivalent to specifying the three web server hostnames with the `-H` or `--hosts` flag.

The previous example relies on a statically-defined group of hostnames, which is obviously not suitable for the dynamic nature of EC2. Having to manually create a list of web instances in the production environment before each deployment would quickly become tiresome. Fortunately, role definitions can be set dynamically: before each task is executed, it checks the role definitions to get a list of hostnames. This means one task can update the role definitions, and the updated definitions will be used for all subsequent tasks.

We can use this feature to create different role definitions for staging and production environments. Suppose we are running an infrastructure using hostnames that reference the server's role and environment. For example, a web server in the production environment is named *www1-prod*. Example 7-1 shows how dynamic role definitions can be used to control which hosts the tasks are executed on.

Example 7-1. Dynamic Fabric Role Definitions

```
from fabric.api import env

def production():
    env.roledefs = {
        'web': ['www1-prod', 'www2-prod', 'www3-prod'],
        'db': ['db1-prod', 'db2-prod']
    }

def staging():
    env.roledefs = {
        'web': ['www1-staging', 'www2-staging'],
        'db': ['db1-staging']
    }

def deploy():
    run('deploy.py')
```

Remember that Fabric tasks are simply Python functions—they do not necessarily need to execute any code on remote servers. We could then update the staging servers with the following command:

```
fab staging deploy
```

This command makes Fabric execute the `staging` task to set the role definitions, then run the *deploy.py* script on the remote instance.

This example runs the non-existent *deploy.py* script, which acts as a placeholder for your own deploy script. Later we will create a *deploy.py* script for the example application from the previous chapter.

That works well for situations where we know all of our hostnames in advance, but how about dynamic fleets of EC2 instances where we don't even know how many instances there are, let alone their hostnames?

In combination with the EC2 API, we can take advantage of this feature to selectively execute tasks on our EC2 instances without needing to know the hostnames in advance. This relies on the tagging strategy introduced in the preceding chapters, where each instance is tagged with a role and an environment. Instead of setting env.roledefs to a list of pre-defined hostnames, we will query the EC2 API to find a list of instances that match our target role and environment.

Dynamically finding instances

I have released an open-source package encapsulating the logic required to query the EC2 API and use tags to build up a list of EC2 instance hostnames. This Python module can be used to quickly convert the previous example—which showed how to deploy software to different environments—into a script that can be used to deploy code to all running instances known to EC2.

Example 7-2 uses the EC2 Tags feature. It assumes that each instance was created with a "web" or "db" tag and puts the hostnames into the associated key of an associative array called roles. For each environment we need (production, staging, and deploy) we reed the associative array into our environment.

Example 7-2. Fabric Role Definitions and EC2 Tags

```
from fabric.api import run, sudo, env
from fabric_ec2 import EC2TagManager

def configure_roles(environment):
    """ Set up the Fabric env.roledefs, using the correct roles for the given environment
    """
    tags = EC2TagManager(AWS_KEY, AWS_SECRET,
        regions=['eu-west-1'])

    roles = {}
    for role in ['web', 'db']:
        roles[role] = tags.get_instances(role=role, environment=environment)

    return roles

def production():
```

```
    env.roledefs = configure_roles('production')

def staging():
    env.roledefs = configure_roles('staging')

def deploy():
    run('deploy.py')
```

This can be executed in exactly the same way as the previous static example:

```
fab staging deploy
```

Assuming you have some running instances bearing the relevant tags—a "host" tag with a value of "web" or "db", and an environment tag with the value "production"—the deployment task will be executed on each of the matching EC2 hosts.

If you wanted to run the deployment task only on the DB instances, as an example, you could execute:

```
fab staging deploy --roles db
```

The tags given are just examples. Any key/value pairs can be used with EC2 tags and queried from Fabric, making this a flexible method of orchestrating your EC2 fleet.

Updating Instances at Launch Time

The second part of the problem is to update newly-launched instances. If an AMI has a particular version of an application baked into it, that is the version that will be running when the instance is launched. If a new version has been released since the AMI was created, the instances will be running an outdated version.

Our instances therefore need to be able to check for the latest version as part of the boot process, and perform an update themselves if necessary. All operating systems provide some mechanism for running user-defined scripts at boot time. For example, on Linux systems, the update can be triggered by placing the following in the */etc/rc.local* file:

```
#!/bin/bash

/usr/local/bin/deploy.py
```

In this example, the deploy script would need to check a central location to find the "latest" version of the application and compare it with the currently-installed version. The script could then update the instance to the correct version if a change is required.

Package Management

Many programming languages and operating systems offer their own solutions for distributing code and configuration files, such as PyPi for Python, Yum for RedHat-based systems, and Apt for Debian-based systems. Using these systems where possible can

make for a very easy upgrade path, because you can rely on other aspects of the ecosystem to reduce the amount of work you need to do.

For example, Python's distribution systems provide a requirements text file that lists all of the Python modules required by your application. The requirements file—commonly named *requirements.txt*—also tracks the installed version of the package. Moving from Boto version 1.1 to 1.5 requires a single change in requirements.txt (from "boto==1.1" to "boto==1.5").

If you package your Python code as a module and publish it to an internal PyPi repository, you can then deploy your application by changing the requirements file used by your running instances.

Another option is to build operating system packages (e.g., RPM packages for RedHat systems) for your custom application so that they can be installed with the OS's own package management system, and host these on your own repository server. EC2 instances can simply check this repository for any updated packages when they are launched, ensuring that they are always running the correct version of each software package.

Building such systems is not within the scope of this book, but they can be very useful for managing AWS-hosted applications, and are well worth the time required to learn and implement.

AMI-based Deployments

AMIs are the building blocks of EC2. Deploying AMIs instead of individual code or configuration changes means you can offload the work of replacing instance to Amazon with varying degrees of automation.

Deploying AMIs with CloudFormation

The most automatic method involves using CloudFormation to replace running instances by changing the AMI ID referenced in the stack template. When CloudFormation receives an update request with a new AMI, it will launch some instances running the new AMI and then terminate the old instances. The number of instances launched will match the number of currently running instances to avoid a sudden reduction in capacity.

Using this method requires a high degree of confidence in the new AMI. You should first test it thoroughly in a staging environment, perhaps with an automated suite of test cases. If the deployment needs to be reverted—going back to the previous AMI—your application will be unavailable until CloudFormation has finished replacing the instances once more.

Furthermore, once CloudFormation starts processing an update, there is no alternative but to wait for it to finish. If you discover early on in the update process that the application is not working as expected, you will need to wait for CloudFormation to finish applying the broken update before issuing a command to perform another update to revert to the previous AMI.

> Netflix, a long-term heavy user of AWS, has released Asgard, described as a web-based Cloud Management and Deployment tool. This can be used to automate deployments and changes to your infrastructure and control the process from a web interface. It acts as a supplement to the AWS Web Console and is more tightly-integrated with Netflix's deploy process.

Deploying AMIs with the EC2 API

An alternative approach is to automate the replacement process yourself using the EC2 API. Instead of allowing CloudFormation to update the running instances, you use the API to perform the same process from your own script. This gives you the opportunity to insert checkpoints and handle the rollback or reversion process with more granularity.

Using the EC2 API opens up some opportunities that not available in traditional environments. Consider an application that has two web server instances running behind an Elastic Load Balancer, running version 1 of the AMI. The update process could then perform these steps:

1. Launch two instances running version 2 of the AMI.
2. Wait for these instances to be ready.
3. Query the health check page until the instance is serving requests correctly.
4. Add the new instances to the ELB, and wait for them to begin receiving traffic
5. Remove the old instances from the ELB.

At this point, the old instances are still running but not actually serving any traffic. If version 2 of the AMI turns out to be broken, you will see an increase in failed requests as measured by the ELB. At this point you can quickly revert to version 1 of the AMI by performing the update process in reverse—adding the old instances to the ELB and removing the newer ones.

Once you are sure the new version of the AMI is working properly, the old instances can be terminated.

Recap

The best choice will depend on how frequently you deploy code changes, and how much work is required to create an AMI. Updating running instances is preferable when you are frequently deploying small changes, and you need new code to be live as soon as possible.

Building new AMIs can provide a cleaner way of deploying updates. The new AMI can be thoroughly tested in a staging environment before it is deployed.

The process of deploying an update can be reduced to simply changing the AMI ID used in a CloudFormation stack, and the task of replacing running instances is handled by Amazon's internal systems. This approach opens up some interesting methods that really take advantage of the cloud and the temporary nature of EC2 instances. Some basic principles to take away from this chapter are:

- Automate the process of building new AMIs as soon as possible, to reduce "deploy friction". The easier it is to deploy changes, the faster you will be able to iterate.
- Do not wait until you have deployed a broken update to start thinking about how to revert.
- AWS has some advantage over physical hardware, such as launching new instances instead of updating code on running instances. Using the EC2 APIs creatively can save both time and headaches.

Building Reusable Components

Introduction

It is the goal of any time-pressed systems administrator to avoid duplication of work where possible. There is no need to spend time building ten servers when you can build one and clone it, or implement a configuration management system that can configure ten servers as easily as one.

Within the context of AWS, there are a number of ways to work smarter instead of harder. Remember that AWS gives you the building blocks you need to build your infrastructure. Some of these blocks can be reused in interesting ways to remove a number of tedious steps from your workflow.

As an example, consider an application that runs in three environments: development, staging, and production. Although the environments differ in some important ways, there will definitely be a lot of overlap in terms of the AWS resources required and the application's configuration. Considering the reuse of resources will save a lot of time as your infrastructure grows, and will let you take advantage of the flexibility that makes cloud hosting so useful.

This chapter looks at some of the ways in which AWS components can be designed for optimal reusability in order to reduce development time and reduce time spent on operations and maintenance.

Role-based AMIs

It's common in both AWS infrastructures and traditional datacenters to assign each instance or server a role that describes the functions it will perform. A web application consists of multiple roles: serving web requests, processing asynchronous tasks, providing a database and so on.

The most popular configuration management tools provide some method of implementing a role-based architecture. In fact, it might be said that the *raison d'être* of configuration management tools is to provide a way to assign a role to a server or virtual instance. Applying a set of Puppet modules (or Chef recipes, or Ansible playbooks) to an instance prepares it to perform its role.

The speed with which cloud computing allows you to bring new instances online makes it even more feasible to adopt this approach fully and design your infrastructure so that each instance performs one role, and one role only. When it took days or weeks to bring a new server online, it was much more tempting to add "just one more" role to an already over-burdened server.

This raises the question of how AMIs can be used to facilitate this approach. If an instance performs only a single role, do you need one AMI per role? Not necessarily. It is possible to create an AMI which can perform multiple roles, as we saw in Chapter .

When the instance is launched, the configuration management tool will be run. This *launch-time configuration* transitions the instance from the *launch state* to the *configured state*, at which point it should be ready to begin performing its role.

Using a single AMI for all roles means that each instance launched from this AMI will need to perform a lot of role-specific configuration at launch time. An AMI should always contain just the bare minimum number of installed software packages that are required by its instances. In this architecture, the AMI will need the packages required by each and every role. This leads to a lengthy launch-time configuration process, that will increase the amount of time it takes for an instance to be ready to perform its regular duties.

At the other end of the spectrum, creating an individual AMI for each role results in a much shorter launch-time configuration process, although at the cost of an increase in time spent creating and managing AMIs.

Making and testing AMIs is not difficult, but it is time-consuming. If you are using an automated AMI creation process, this cost becomes a lot easier to bear.

There is a third option that involves a different approach to building a base AMI that can be reused for all roles. With this method, the base AMI contains all software packages required to perform any of the roles in your infrastructure. For example it might include packages for a database server, web server, and in-memory caching server. However, none of the services are configured to start when the AMI is launched. Instead, the configuration management tool takes responsibility for starting services. This reduces the amount of time taken to perform the launch-time configuration, because software packages do not need to be downloaded and installed, only configured and started.

One downside of this third approach is that the base AMI might need to be rebuilt more frequently than role-specific AMIs. If your base AMI includes both PostgreSQL and

Nginx, the release of an urgent update to either package will necessitate rebuilding the AMI and replacing any instances that are running the old version. Failing to do so would result in running insecure versions of core software, or result in running two versions of the base AMI, which will quickly become inconvenient to manage.

No matter which method you choose, you will need some way of mapping a running instance to a particular role, which is the topic of the next section.

Mapping Instances to Roles

Once you know that each instance will perform a given role, how do you apply that role to the instance?

The answer to this question comes in two parts: assigning roles in AWS, and making this information available to your configuration management system. The second task is covered in the next section.

Amazon offers two main ways to assign roles to instances or, looking at the problem from a higher level, pass arbitrary information to your instances: User Data and Tags. Of course it is also possible to store such information in a database such as RDS, SimpleDB, or your own database.

User Data is usually the easiest method for a number of reasons. Instances can access User Data without needing any IAM access credentials. User Data can be retrieved from a static URL, which makes it usable from almost any programming language.

The most basic method of using User Data to control role assignation would be use the entire User Data field to specify the role. This value would then be available to scripts running on the instance, for example:

```
#!/bin/bash

ROLE=$(curl http://169.254.169.254/latest/user-data)
echo My role is $ROLE
```

In practice, you might already be using User Data for something else and not want to use the entire field for the role. In this case, you should mov to a more suitable data structure, such as JSON or key/value pairs.

Using Tags is slightly more complicated because they can be retrieved only from the AWS command line tools or APIs, the latter requiring the use of a programming language with a suitable AWS client library. An IAM account must be used to access the APIs, which means that any use of Tags requires access to a set of IAM access credentials.

The simplest way to provide these credentials is to use IAM Roles so that credentials do not need to be hard-coded into scripts or retrieved from User Data. Instead they can be automatically detected by the underlying client library used by your scripts (such as Boto, introduced in "Launching From Your Own Programs and Scripts" on page 22).

Tags have three advantages over User Data. First, they are *key=value* pairs by design. This fits in very neatly with the idea of mapping a role to an instance: We simply need to create a tag named `role` and give it a value such as `web-server` or `db-server`.

The second advantage of Tags is that they can be queried from outside of the instance far more easily than User Data. Tags make it possible to perform an API query to retrieve a list of instances tagged with a role such as `web-server`. Performing the same query for User Data would involve listing all of your instances, parsing the User Data, and compiling a list of matching instances.

The final advantage of Tags is more of a business reason than a technical one. Tags can be used by Amazon's billing system to produce an itemized list of instances divided into groups based on the arbitrary tags you have defined in your Cost Allocation Report. The same source of data informs both technical and business decisions.

Example 8-1 shows an example of retrieving Tags from a Python script.

Example 8-1. Using EC2 Tags

```
from boto.utils import get_instance_metadata

metadata = get_instance_metadata()
my_instance_id = metadata['instance_id']


conn = boto.ec2.connect_to_region("us-west-1")
reservations = conn.get_all_instances(filters={'instance-id': my_instance_id})
instances = [i for r in reservations for i in r.instances]

# instance-id is a unique identifier so it is safe to assume there is only one instance
instance = instances[0]

# Iterate through the tags, printing the keys and values
for key, value in instance.tags:
    print "Key is %s. Value is %s" % (key, value)
```

This script does not include any IAM credentials. It assumes that the instance it is running on has been assigned an IAM Role. Boto will automatically use IAM Role credentials if they are available.

Patterns for Configuration Management Tools

As mentioned earlier, reusability is a core goal for many configuration management tools, the entire purpose of which is to reduce duplication of effort by automating common tasks such as creating files or installing software packages.

This section will look at how this role-based design pattern can be used within Puppet, building on the information in the previous section. Note that, apart from the syntax

used, there is nothing specific to Puppet about this pattern. It can be implemented in all the configuration management tools of which I am aware.

The usual *modus operandi* of Puppet is to use the hostname of an instance to control which configurations are applied to that instance. This is practically useless within AWS, because hostnames are automatically generated based on the IP address of the instance. It is possible to work around this by setting the hostname to a "useful" value before running Puppet—a valid tactic that is used by companies such as Pinterest.

In our case, however, we want to bypass hostnames completely and use the role attribute that we assigned by way of User Data or EC2 tags.

To find out information about the environment in which it is running, Puppet depends on a tool named Facter. Facter is responsible for providing Puppet with "facts" about the system it is running on. These facts can then be used within your Puppet modules. The instance's hostname is a fact. Puppet modules make this available via the $HOST-NAME variable.

Facter has built-in support for EC2, which means that it will automatically provide certain EC2-specific data to Puppet. Facter will actually query all of the available Meta and User Data variables and provide them as facts to Puppet. For example, the AMI ID is available at a URL with a structure like *http://169.254.169.254/latest/meta-data/ami-id*. Facter will automatically set the `$ec2_ami_id` fact to this value. Note that the variable name is prefixed with `EC2_`, and any dashes are replaced with underscores.

For the sake of our example, let us assume that we are using JSON-formatted User Data with our instances, and our web server has been launched with the following User Data:

```
{ "role": "web",
  "environment": "dev" }
```

This JSON object declares two attributes: a role and an environment.

Facter populates the `$ec2_user_data` variable to make this information available immediately to Puppet. However, Facter does not know that this data is JSON formatted: as far as it is concerned, the User Data is simply a string of text.

Puppet's stdlib module provides the `parsejson` function you need to extract the keys and values you need from the JSON object. The function converts a given text string into a JSON object and returns the result as a hash. Once the data is in a hash, you can access the role and environment attributes and use them in conditional statements within Puppet modules, as shown in Example 8-2.

Example 8-2. User Data Roles and Puppet

```
node default {

    require stdlib
```

```
    $userdata = parsejson($ec2_userdata)
    $role = $userdata['role']

    case $role {
        'web': {
            require role::www::dev

        }
        'db': {
            require role::db::dev
        }
        default: { fail("Unrecognized role: ${role}") }
    }

}
```

This example shows how the `$role` attribute can be used to control which modules are applied to each instance. The `$userdata['environment']` variable could be used to provide a further level of abstraction, so that the live environment uses the `role::www::live` module, and the development environment uses `role::www::dev`.

For the sake of brevity, I have not included the www and db Puppet modules. These are simply Puppet modules that perform tasks such as installing Nginx or PostgreSQL.

> The module layout—in this case, `role::www::dev` and `role::db::dev`—is based on Craig Dunn's Design Puppet → Roles and Profiles blog post. This is a great way to separate business logic ("What should this instance be doing?") from technical details ("How should this instance be configured?"), and is particularly useful when adopting this pattern in AWS.

User Data is only one way of providing information to AWS instances so that it can be made available to Puppet. The other option is to create Tags on the instance and make these available to Puppet.

Unfortunately, Tags support is not built in to Facter as easily as User Data. This is a minor hurdle to bypass, though—Facter makes it easy to add additional facts by way of a plugin architecture. Plugins are simply Ruby scripts placed in a particular directory, where the name of the script is also the name of the fact that will be returned. Facter executes all of the plugins while gathering facts about the system.

Example 8-3 shows an example Facter plugin that retrieves all of the Tags for the instance and make them available to Puppet.

Example 8-3. EC2 Tag Facts

```
require 'facter'
require 'json'
```

```
if Facter.value("ec2_instance_id") != nil
 instance_id = Facter.value("ec2_instance_id")
 region = Facter.value("ec2_placement_availability_zone")[0..-2]

 cmd = <<eos
  aws ec2 describe-tags
   --filters \"name=resource-id,values=#{instance_id}\"
   --region #{region}
   | jq '[.Tags[] | {key: .Key, value: .Value}]'
 eos
 tags = Facter::Util::Resolution.exec(cmd)

 parsed_tags = JSON.parse(tags)
 parsed_tags.each do |tag|
  fact = "ec2_tag_#{tag["key"]}"
  Facter.add(fact) { setcode { tag["value"] } }
 end
end
```

For more information about adding custom facts to Facter, and to find out where on your system this plugin should be located, see the Custom Facts documentation page.

With this plugin in place, we can launch an instance and assign role and environment Tags to it, instead of passing this information as User Data. The code shown earlier in Example 8-2 has to be modified so that, instead of parsing JSON from the `$ec2_user data` variable, it obtains the same information by using the `$ec2_tag_role` and `$ec2_tag_environment` variables, for example:

```
node default {

    case $ec2_tag_role {
        'web': {
            require role::www::dev
    ...
```

Although this section has focused on Puppet, the same result can be achieved with most other configuration management tools. The general principle of providing some information to the instance at launch time, and then using this information later to control the instance configuration, can be used from configuration management tools or your own scripts.

Modular CloudFormation Stacks

CloudFormation stacks can also be designed to make them more suitable for reuse in different parts of your application. This section looks at one of the most popular method of reaching this goal.

This method involves the use of the `AWS::CloudFormation::Stack` resource, which lets you embed one CloudFormation template within another. That is, a "parent" Cloud-Formation stack can create a number of "child" stacks. The parent stack can provide Input values to the child stack and access its Output values. This means that the parent stack can create multiple child stacks, and use the Outputs of one stack to as the Parameters of another stack. In this design pattern, each child stack can be self-contained and highly focused on a particular task, such as creating EC2 instances or an RDS database.

> To use embedded templates, you will need to provide a `Templa teURL` parameter, which tells CloudFormation where to download the stack template file. This file must be in an S3 bucket configured to serve its contents over HTTP. For more information see the AWS Stack Properties documentation.

The parent stack is responsible for tying all of these components together and providing the foundation your application needs to run. This is illustrated in Figure 8-1.
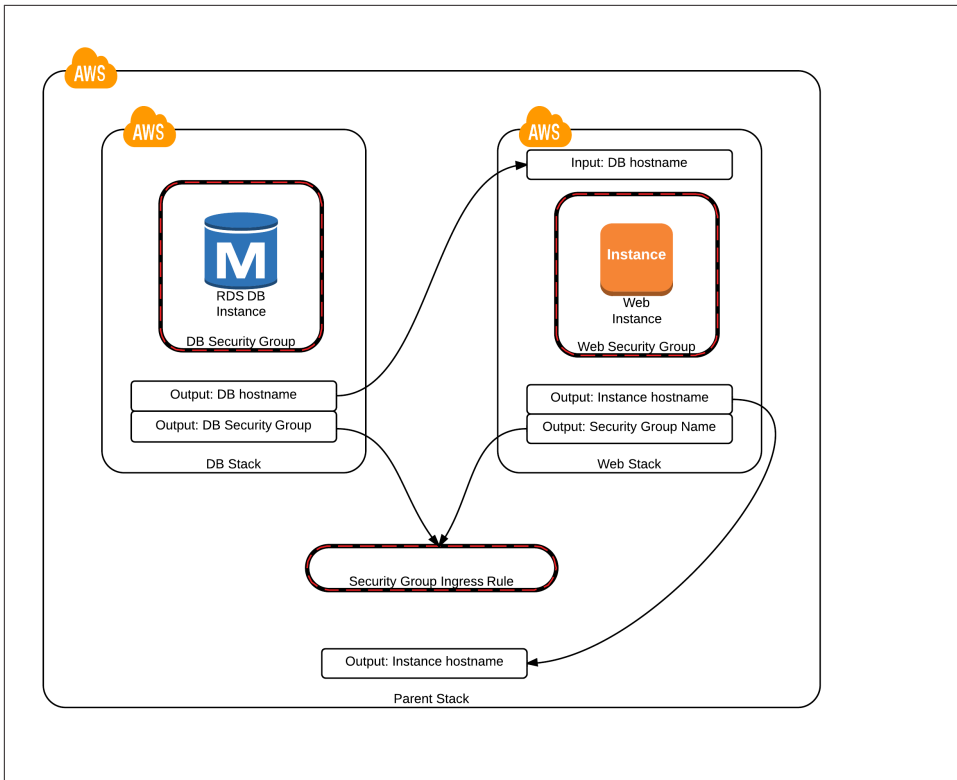
Figure 8-1. Embedded CloudFormation Stacks

The architecture in the figure consists of three CloudFormation stacks: the parent stack, the DB stack, and the web stack. The DB stack is responsible for creating an RDS instance and placing it in a Security Group. The Web stack creates an EC2 instance, also in a Security Group.

When the RDS instance is created, it is assigned a unique hostname generated by Amazon, which cannot be predicted in advance. How then can you let the instance know the hostname of the database instance so that it knows where to send data requests? The answer comes in the form of Parameters and Outputs. These can be used to provide data when launching a stack, and t retrieve dynamic stack attributes once it has been created.

In this case, the DB stack outputs the hostname of the RDS instance. The parent stack uses this value as an Input when creating the Web stack. In turn, the Web stack passes this value to the instance as User Data or a Tag, so that it can be used in your configuration management software. More information on using Parameters and Outputs with embedded stacks can be found on Amazon's Stack Resource Snippets page.

CloudFormation attempts to automatically determine which order the resources should be created in, so that Parameters and Outputs can be passed around successfully. In this example CloudFormation would detect that the Output from the DB stack is used as an Input to the Web stack, and would therefore create the DB stack first.

> If you need more control over the order in which CloudFormation resources are created you can use the DependsOn attribute to override CloudFormation's automatic ordering.

Note that neither the Web nor DB stacks create any Security Group rules. Instead, these are created in the parent stack through the use of the `AWS::EC2::SecurityGroupIngress` resource type.

This resource allows you to create Security Group ingress rules and assign them to existing Security Groups. Why are these created in the parent stack? This is a personal preference more than a technical requirement. It would be possible, for example, to pass the DB Security Group ID as an Input to the Web stack and allow the Web stack to create the necessary ingress rules, permitting the instance to access the database.

But the opposite is not possible: in order to pass the Web Security Group as an Input to the DB stack you would need to create the Web stack before the DB stack. That would make it impossible to provide the DB hostname as an Input to the Web stack, meaning the instance would not know the address of the database server.

By creating the ingress rules in the parent container, you are simplifying the DB and Web stacks. They should not be concerned with creating Security Group rules, as it can be argued that ingress rules are not specifically tied to the function of the stack.

Moving some of the resource creation—such as ingress rules—to the parent stack increases the reusability of the child stacks. Consider the familiar example of development and production environments. Both will need Web and DB stacks, but the development instance should be accessible only from a specific list of IP addresses, whereas the production environment will be accessible from the public Internet.

To enforce this distinction, you could create a separate CloudFormation stack for each environment, each of which embeds the Web and DB stacks as children. The development and production stacks would be almost identical, except when it comes to creating the Security Group ingress rules. When you want to make changes to the DB stack, you have only a single template to update.

This provides a clean way of breaking your infrastructure into logical components that can be reused to create flexible environments without duplication of effort.

CloudFormation will automatically tag your resources with some information about the stack itself. If you configure Cost Allocation Reports to track these tags, you get an extremely high level overview of where your money is going.

A number of other tricks can be used in CloudFormation templates to allow easy recycling of stacks. Let us suppose that we need each developer on the team to have their own development environment, which should be accessible via SSH at a given hostname such as `mryan.dev.example.com`.

The development stack template could accept an EnvironmentName Input parameter, which is used to create a Route 53 resource record mapping the desired hostname to the instance's public DNS name. Each developer can create his own copy of the stack using the same template, entering their username when the stack is launched.

Although the CloudFormation template language is not as flexible as a full programming or scripting language, it can be useful to think of your CloudFormation stacks in the same way that you think of your scripts, programs, or even Puppet modules. Each of these presents various methods that can be used to dramatically cut down how much time it takes to add new features or deploy new resources.

Log Management

Logging in the Cloud

Despite the best efforts of system administrators everywhere, logging in the cloud can quickly become more complicated (and more expensive) than logging in a phsyical hardware environment. Because EC2 instances come and go dynamically, the number of instances producing log files can grow and shrink at any time. Your logging system must therefore be designed with this in mind, to ensure it keeps up with peaks in demand when processing log files.

Another area that requires some forward thinking is log storage. Running a large number of instances will produce large log files, which need to be stored somewhere. Without some advance planning, the storage requirements can grow rapidly, leading to an increase in costs.

This chapter looks at some popular logging tools that can be useful in AWS environments and introduces some stategies for managing log files without breaking the bank. Logstash is used to demonstrate the concepts in this chapter, but the principles will also apply to most other logging software.

Central Logging

A common solution to the problem of viewing logs from multiple machines is to set up a central logging server to which all servers in your infrastructure send their logs. The central logging server is responsible for parsing and processing log files and managing the policies for log retention.

This pattern works well within AWS, with a few caveats. As mentioned earlier, it is critical to ensure that your logging system does not struggle to keep up when there are many instances sending their log files to the central server.

Another potential issue is related to hostnames within EC2. Remember that, by default, most instances will set their hostname to match their internal IP address. This is not particularly useful when it comes to viewing log files. For this reason, many log-viewing tools provide a method of adding key/value pairs to log data. These can be used in conjunction with EC2 Tags to make it easier to keep track of the source of a particular log entry.

There are three main components to building a central logging system. Log shippers and log receivers such as syslog, rsylog, and syslog-ng are responsible for sending and receiving log files. The third component is log viewers, such as Kibana and Graylog2, which handle the task of displaying this gathered data through a web interface. To further complicate things, most log shippers can also act as log receivers, and some packages provide all three components.

Unfortunately, a comparison of the various tools that can be used to build such a system is beyond the scope of this book, because the issues surrounding them are not really specific to AWS. However one tool which deserves special mention, because it has a few AWS-specific features.

Logstash is an open source log management tool that provides a framework for receiving, processing and storing logs. It also includes a web interface for browsing logged data.

Like many log receiving tools, it can accept data in the standard syslog format (RFC 3164). It also has a plugin architecture that allows it to consume log data from a variety of additional sources. Most interestingly, it can read and write logs stored in S3 buckets, read logs from SQS Queues, and write output messages to SNS Topics.

This integration with AWS services makes Logstash an ideal candidate for building a logging system within AWS.

Logstash itself consists of multiple components. It includes the core functionality of consuming and producing log files, as well as a built-in web interface (Kibana). It also comes with a built-in instance of ElasticSearch, which is a distributed search server based on Apache Lucene. ElasticSearch provides search capabilities allowing you to quickly and easily find the log entries you are searching for.

All of these components can be run on a single machine for development and testing purposes. Once Logstash is dealing with a large amount of log data these components can be moved to separate instances so they can be scaled independently.

Unfortunately, Logstash is still relatively new. As a result there is limited support for Logstash in popular package managers—it is not currently possible to install Logstash with a simple *apt-get install logstash*.

Of course, there are also many third-party logging services that can entirely obviate the need to build your own logging system. Many of these services provide some convenient

features such as automatically retrieving EC2 tags and assigning them to log data, so that EC2 tags can be used to quickly drill-down through log files.

Logstash Configuration

To demonstrate Logstash in action, we will set up a simple centralized logging infrastructure suitable for use in EC2. Before going ahead and setting up the instances, we will need to first prepare the Security Groups that will be used in the demonstration.

> For the sake of keeping the demonstration simple, we will manually install and configure Logstash on the client and server instances. Of course, when it comes to moving this into production, the configuration should be handled by a tool such as Puppet. The Logstash documentation site contains links to Logstash Puppet modules that can be used to install and configure the Logstash components.

Using the Web Console or command-line tools, create two Security Groups, named log_client and log_receiver. The rules for the log_client can be left empty.

The log_receiver Security Group will be used for the Logstash server. It must be able to accept syslog traffic from logging clients, and allow administrators to access the Kibana web interface and ElasticSearch API.

Create four rules in the log_receiver group:

Rule	Description
Inbound TCP 9292 from your network	Kibana web interface
Inbound TCP 9300 from your network	ElasticSearch API
Inbound TCP 5514 from log_client Security Group	Syslog
Inbound TCP 22 from your network	SSH

Create a single rule in the log_client group:

Rule	Description
Inbound TCP 22 from your network	SSH

Once you save these changes the Security Group configuration is complete.

Creating and configuring a Logstash server

After creating the Security Groups, you can launch and configure the Logstash server instance. Launch a new EC2 instance using the Ubuntu LTS AMI (or your favorite distribution). Optionally use Route 53 to set up a DNS record for this instance, so that `logging.example.com` points to the public DNS name of the new instance.

When the instance is ready, connect to it via SSH and download the Logstash application:

```
wget http://logstash.objects.dreamhost.com/release/logstash-1.2.1-flatjar.jar
```

Ensure you have a recent version of Java:

```
apt-get -y install openjdk-7-jre-headless
```

Logstash is configured by way of a configuration file, the path to which is specified when launching Logstash. Create a file named *logstash-central.conf* containing the following contents:

```
input {
  tcp {
    type => syslog
    port => 5000
  }
}

output {
  stdout { codec => rubydebug }
  elasticsearch { embedded => true }
}
```

The Logstash config file consists of three sections, two of which are shown here. The `input` section specifies sources of logging data which Logstash will consume. The `out put` section controls what Logstash does with this data once it has been filtered. The final section (`filter`) is not required for this simple setup. Filters are used to control the flow of messages and to add supplementary data to log entries, and are not used here.

> For more information on using filters to parse and modify syslog data, see the Logstash documentation.

In this case, all data is output to an instance of ElasticSearch as well as stdout. We output to stdout so we can easily see logged messages on the console. The use of the console is for development purposes only and should be removed when moving in to production.

The `embedded => true` parameter tells Logstash to use an embedded instance of ElasticSearch. That is, Logstash will take care of launching ElasticSearch and forwarding log data to it.

With the configuration file saved, launch Logstash with the following command:

```
java -jar logstash-1.2.1-flatjar.jar agent -f logstash-simple.conf -- web
```

This command will launch the main Logstash process and the additional processes required to run the Kibana web interface (as denoted by the final two `--web` arguments). Thus, we are effectively running two commands simultanously here: one to launch the Logstash agent, and another to launch the Logstash web interface.

Launching Logstash might take some time, as it needs to extract the jar file to the working directory before launching the processes required for the Logstash agent and web interface, as well as the embedded ElasticSearch instance. Once Logstash has launched, you should see something similar to the following:

```
< logstash startup output >
```

In another SSH session, verify that Logstash is listening for incoming syslog data on TCP port 5000 by using the *netstat* command:

```
root@ip-10-80-1-109:~# netstat -anlp | grep 5000
tcp6      0      0 :::5000                 :::*                    LISTEN      6544/java
```

Finally, use the *netcat* command to send a test message to Logstash. This message should be printed in the terminal session in which you started the Logstash agent:

```
echo "testing logging" | nc localhost 5000
```

Once this command is executed, you should see some output printed to the Logstash console. This is a JSON representation of the logged message, for example:

```
{
        "message" => "testing logging",
     "@timestamp" => "2013-09-09T09:10:51.402Z",
       "@version" => "1",
           "type" => "syslog",
           "host" => "127.0.0.1:35804"
}
```

Finally, browse to the Kibana web interface with your web browser. If you set up a Route 53 record to point *logging.example.com* to your instance, you can visit the interface at *http://logging.example.com:9292/*. Otherwise, use the public DNS name of your EC2 instance, such as *http://ec2-54-217-126-48.eu-west-1.compute.amazonaws.com:9292/*.

Using Kibana's search function, search for the string "testing". The results window should then show you the test message you logged with the logger program.

Configuring the Logstash clients

With a server listening, you can move on to configuring the logging client that will represent the other servers in your infrastructure.

Launch a second EC2 instance, and make it a member of the log_client Security Group. Once the instance is ready to accept SSH connections, you can log in to download the Logstash agent and install Java:

```
wget http://logstash.objects.dreamhost.com/release/logstash-1.2.1-flatjar.jar
apt-get -y install openjdk-7-jre-headless
```

The Logstash client configuration will differ slightly from the central Logstash configuration. Instead of outputting your log files to ElasticSearch, you will instead send them to the central Logstash instance using the syslog protocol.

Create a client configuration file named *logstash-client.conf* with the following contents:

```
input {
  file {
    type => "syslog"
    path => [ "/var/log/*.log", "/var/log/syslog" ]
  }
}

output {
  stdout { codec => rubydebug }
  tcp {
    host => "logging.example.com"
    port => 5000
  }
}
```

As before, this configuration file defines the inputs and outputs that Logstash will use. The input in this example is a list of files. Logstash will read these files and ingest log entries as they are written. The path attribute accepts a list of paths, which can either be glob paths (i.e., including an asterisk or other wildcard) or paths to single files, as in the case of */var/log/syslog*.

> If you did not set up a Route 53 DNS record for your central log server, you will need to add the instance's public DNS name to the client configuration file, instead of logging.example.com.

Therefore, this configuration file will cause Logstash to monitor the specified files and send their input via TCP to the *logging.example.com* host, where syslog is listening on port 5000. We are not yet using any filters to modify or parse the logged data.

On the client server, start the Logstash agent with:

```
java -jar logstash-1.2.1-flatjar.jar agent -f logstash-client.conf
```

Use the logger command to write some example text to the local syslog system:

```
echo "testing client logging" | logger
```

If everything is configured correctly, you should see the test message repeated in the logstash receiver console. In addition, this message will be passed to ElasticSearch, so it can also be viewed by searching for "testing" in the Kibana web interface.

With those steps complete, the basic central logging system is up and running. Any log messages produced on the client system and written to one of the monitored files will be passed to the central Logstash server.

Logging to S3

Like any critical component, the logging system should be loosely coupled to the other moving parts in your infrastructure. That is, a failure in the logging system should not propagate and cause other services to fail.

In the previous example, we set up a central Logstash server that will accept log messages from clients via TCP. What happens if the central Logstash instance crashes, or otherwise becomes unavailable? In that case, clients will be unable to send log messages. Any attempt to do so would result in an "broken pipe" error message, because the client is unable to open a TCP connection to the central server. Fortunately, the Logstash client will recognize this failure and spool the log messages locally. Once the central Logstash server is back in action, these spooled messages will be resent.

> If we were using UDP, transmissions would actually fail silently and messages will be lost instead of being stored at the senders. This may or may not be acceptable, depending on your log retention policies.

This process of storing the messages locally until they can be sent to the master is known as "store and forward", and is similar to the way in which systems like email work. If the Logstash agent is only temporarily unavailable, storing the messages temporarily on the client will not cause any problems. However, prolonged outages might cause an excessive amount of spooled data to pile up on the client instance. This can cause problems if the temporary files grow so large that they interfere with the proper running of the client instance.

In this case, it can be helpful to have an intermediary storage location for your log files, to further decouple the client/master Logstash instances. One method of doing this is to use S3 as temporary storage for your log files: instead sending its log files directly to the central Logstash server, the client writes all log files to an S3 bucket.

The central Logstash agent is then responsible for regularly downloading these log files from S3 and processing them as usual.

The S3 intermediary has a number of benefits, the primary one being decoupling. Even if the central Logstash is unavailable for an extended period of time, you can be safe in the knowledge that your log files will be queued up on S3 and processed once the central log server is back in action.

A secondary benefit relates to scaling your logging system. Consider what would happen if the number of instances sending their log files to the central Logstash instance were to grow rapidly. Because log messages are sent to Logstash as soon as they are generated, logs are effectively processed in real time. Sending too many logs could cause the central instance to become overloaded.

By temporarily storing the files in S3, you can remove the instanteous nature of the processing. Instead, the central server has more control over when log files are pulled from S3 for processing. While the amount of actual work remains the same, the peaks and troughs are evened out by storing the data on S3 and allowing the central server to pull it.

Because Logstash has built-in support for some Amazon services, including S3, modifying our existing system to support the new setup is extremely straightforward. We need to make two changes to the system. First, instead of clients sending files directly to the central server, they should be written to an S3 bucket. On the central server side of things, we need to tell Logstash to read incoming log files from the S3 bucket, instead of listening for TCP connections.

To begin with, we need to create a new S3 bucket to store our log files. Using the Web Console or command-line tools, create a new bucket with a name like `logs-example-com`. Remember that S3 bucket names must be unique, so you will not be able to use this exact example.

Once the bucket is created, create a new IAM user named *logging*. This will be used on both the client and central Logstash instances to read and write to the bucket.

Assign an IAM policy to the new user, granting it permissions to read from and write to the logging bucket. An example IAM policy is:

```
{
  "Statement": [
    {
      "Action": "s3:*",
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::my-s3-bucket",
        "arn:aws:s3:::my-s3-bucket/*"
      ]
    }
  ]
}
```

Note that this policy explicitly references the S3 bucket and IAM users. You will need to change the example to match the name of your S3 bucket and the ID of your IAM user.

Once the policy has been assigned to the IAM user, the AWS side of the configuration is complete, and you can return to the EC2 instances running your Logstash client and server.

Begin by configuring the Logstash client so that it writes log entries to the S3 bucket. Create a configuration file named logstash-client-s3.conf with the following contents:

```
input {
  file {
    type => "syslog"
    path => [ "/var/log/*.log", "/var/log/syslog" ]
  }
}

output {
  s3 {
    access_key_id => "my-aws-access-key-id"
    secret_access_key => "my-aws-secret-access-key"
    endpoint_region => "eu-west-1"
    bucket => "logging-example-com"
  }
}
```

You will, of course, need to update this example to reflect the name of your S3 bucket and the AWS IAM access credentials.

On the central Logstash instance, create a configuration file named *logstash-central-s3.conf* with the following contents:

```
input {
  s3 {
    bucket => "logging-example-com"
    credentials => ["my-aws-access-key-id", "my-aws-secret-access-key"]
    region => "eu-west-1"
  }
}

output {
  stdout { codec => rubydebug }
  elasticsearch { embedded => true }
}
```

The Logstash S3 extension downloads data from the bucket using th credentials you configure here. So the server's `input` section is totally new in this example, but the `output` section is the same as our original one. Again, you will need to replace the S3 bucket name and IAM access credentials with your own data.

Please note the different configuration file options required for the S3 input and S3 output. Logstash has not yet settled on conventions for naming attributes in plugins, so each plugin author chooses their own variable naming scheme. The documentation for plugins also varies dramatically in terms of quality and professionalism. Hopefully both of these will improve as Logstash continues to grow in popularity.

With these changes in place, you will need to stop and restart the Logstash agents on both the client and central servers. Start the Logstash process on the client with:

```
java -jar logstash-1.2.1-flatjar.jar agent -f logstash-client-s3.conf
```

The central Logstash instance can be started with:

```
java -jar logstash-1.2.1-flatjar.jar agent -f logstash-central-s3.conf
```

Once both processes are running, create some log file entries on the client server with a logger command:

```
echo "testing s3 logger" | logger
```

After a few minutes, this log message should be printed in the central Logstash agent log. This will take a little longer than the preceeding examples, because you will need to wait for the Logstash client to write this message to the S3 bucket, and then wait again for the central Logstash agent to retrieve the updated file and process its contents.

This method of using S3 as a temporary storage location greatly increases the reliability of this logging system, because it is no longer dependent on having the central Logstash agent running. Of course, without the central agent running, log files will not be processed or visible in the Kibana web interface. However, a failure in the central agent will have no effect on client instances, which will happily continue shipping their log files to the temporary S3 bucket for later processing.

AWS Service Logs

So far we have been looking at application and operating system logs, but there is another class of logs that must also be considered. Many of the AWS services produce log files that might need to be stored and reviewed. For example, CloudFront distribution will produce log files providing detail about requests it receives, such as the URL that was requested or the resulting HTTP response code.

All of Amazon's services use the same basic logging methodology: logs are written to a specified S3 bucket at regular intervals. This makes retrieving and processing the log files very simple. You just need to regularly download and process the files. The kind of decoupling described in the previous section is already built in to this system: if you do

not process the log files, they will pile up in the S3 bucket, but CloudFront will continue to function as normal.

Given that we already have a system for processing log files that have been written to an S3 bucket, we can reuse the example from the previous section to read CloudFront logs as well as our application and operating system logs. Logstash is already configured to process logs from an S3 bucket, so we can easily add another section to our central Logstash agent's configuration file to make it process CloudFront log files.

The first step is to create a CloudFront distribution to serve some static or dynamic files, and configure that distribution to store its access logs in an S3 bucket. This is described in Amazon's CloudFront documentation. During this process, you will have the option to create a new S3 bucket in which to store the logs, or enter the name of an existing bucket. Either way, keep track of the bucket name you choose, because this will be required in the following steps. For the demonstration, I have used a bucket named cloudfront-logs-example-com.

Once these steps in the Amazon documentation have been completed, you will have a CloudFront distribution that writes its access logs to an S3 bucket periodically. Next you can configure Logstash to consume these logs, feeding them in to the same system that processes your application and operating system logs.

On the central Logstash instance, stop the *logstash* process if it is still running from the previous example. Modify the *logstash-central-s3.conf* file so that it matches the following:

```
input {
  s3 {
    bucket => "logging-example-com"
    credentials => ["my-aws-access-key-id", "my-aws-secret-access-key"]
    region => "eu-west-1"
  }
  s3 {
    bucket => "cloudfront-logs-example-com"
    credentials => ["my-aws-access-key-id", "my-aws-secret-access-key"]
    region => "eu-west-1"
    type => "cloudfront"
  }
}

output {
  stdout { codec => rubydebug }
  elasticsearch { embedded => true }
}
```

The newly-inserted s3 section configures Logstash so that it will read log files from the CloudFront log bucket, as well as the original logging-example-com bucket.

All logs retrieved from the `cloudfront-logs-example-com` bucket will have their `type` attribute set to `cloudfront`. You can refer to this to keep track of the source of log data, and the type will be visible when these logs are viewed in the Kibana web interface.

After saving the file, start Logstash again with this command:

```
java -jar logstash-1.2.1-flatjar.jar agent -f logstash-central-s3.conf
```

To see this in action you will need to wait for CloudFront to write the first log file, which it will do after receiving a certain number of HTTP requests. For testing, it can be helpful to use cURL to quickly make a large number of requests to your CloudFront distribution, which will cause the access log to be written to. Once this file reaches a certain size, it will be written to S3, at which point Logstash will notice the new file and process the logs contained therein.

Other Amazon services, such as Elastic Beanstalk and S3 itself, use the same mechanism for storing access logs, so this technique can also be reused for those services.

S3 Lifecycle Management

Managing ever-growing log files is an old problem for systems administrators. Working in the AWS cloud introduces some additional challenges but the same principles can be used to solve the problem. On an individual system, logrotate is used to ensure log files are regularly rotated and deleted. Without logrotate, log files might grow to the point where they exhaust all available space on the system, causing problems for running applications.

When storing logs on S3 there is a different problem: instead of worrying about shrinking available capacity you need to worry about an increasing AWS bill. Constantly throwing log files in to S3 buckets and ignoring them will lead to an unnecessarily high bill at the end of the month.

S3 Lifecycles can be used to manage this problem. It allows you to create rules that control when your data is automatically archived to Glacier or permanently deleted. If you are logging to S3 you should ensure that you Lifecycle rules are configured to automatically delete objects when they are no longer required.

Lifecycle rules can also be used to potentially increase the security of your log files after they have been moved to storage. In some scenarios, log files should be considered read-only once they have been written. In strict cases this is enforced using WORM (Write once, read many) drives which provide physical protection to prevent modification of files after they have been written.

While Lifecycle rules can not provide this level of protection they can be used to separate the credentials used for reading and writing operations. For example, image you write your log files to an S3 bucket and they are stored in /backups/logs/. Your logging ap-

plication uses a set of IAM credentials which give it permission to write to this location in the bucket.

You would like to ensure that, once log files have been written, it would be difficult for a malicious user or application to overwrite them. This can be done by configuring Lifecycle rules to `Archive and Delete` the objects after a certain time frame. After this interval the log files would be moved to Glacier, where they would be inaccessible to the IAM credentials used to create the files.

DNS with Route53

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 incredibly simple at its core, a broken or misconfigured DNS server can result in a number of "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.

This book assume that most readers are familiar with the general concepts surrounding DNS. This chapter will therefore focus on the AWS-specific implementation provided by Route 53, and demonstrate 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 very 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, there are some features specific to Route 53 that 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 cotinuously 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 Web 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 restoring 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 it 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). This is a common scenario where a high level of uptime is required, or where a single master is not capable of handling all of the application load on its own. Using tools such as Repmgr, one 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 one 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 recognise the updated DNS records only after being restarted.

This method can be implemented in a number of ways, depending on your infrastructure and the services you are running. In order 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, 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 it is likely that the instances 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.

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
 if slave_rr_exists:  print 'Slave record exists. Adding instance to pool: %s' % slave_hostname else:


def do_promote(): master_rr = zone.get_cname(master_hostname) print 'Updating master record: %s %s' %


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 the AWS_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.

> This script is very 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 which 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 below will be executed on the slave that will be promoted. Note that this script is also responsible for initiating the Repmgr failover process. A script that performs the required actions of initiating the failover process and updating the DNS records is shown in Example 10-2.

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 Web Console. You will see that the *master.example.com* record now points to the instance which 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 began 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 preceeding 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 potentially 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 performant 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 (WRRS), which are used to return multiple records in response to a query for a single DNS hostname.

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

When creating an entry (i.e. a member of the pool) in a WRRS, a weight value must be provided. This is used to calcuate 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 send 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 very 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 non-responsive ELB does not make for a good user experience: users will see a blank, unstyled error page.

Outside of 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 end-point). 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 Web 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 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 which 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 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 Web 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.

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 dropdown, 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 which 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.

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 recognised, 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 everything is configured as described in this section.

Finally, we need to make sure the fail-over 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 recognise 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 honour 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

- Route53 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 Route53 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 Route53 for high availability. Higher TTL values will result in clients using old cached DNS records.

Monitoring

Monitoring dynamic instances can be a challenge. The "classic" monitoring tools expect your instances to be around for a long time, and can have difficulty recognizing the difference between an instance that has failed and an instance that has been terminated as part of an autoscaling event or other planned termination in response to changes in capacity requirements.

AWS provides their own monitoring service, CloudWatch, which has been designed from the ground up to work in such an environment. Additionally, with some planning and custom scripts, most traditional monitoring tools can be used to monitor dynamic instances without spamming operators with false alarms when instances are terminated. This chapter looks at some of these methods, as well as Amazon's CloudWatch service, and how the two can be integrated.

> A cottage industry of cloud-based monitoring tools has sprung up around AWS and other cloud providers. There are too many tools to mention in this book, and each has its own strengths and weaknesses.
>
> The tighter these tools integrate with AWS, the more useful they are. The more advanced tools automatically query the EC2 Tags associated with instances and use them to aggregate metrics. This allows you, for example, to either view a high-level overview of your application across all EC2 regions, or drill down to view the performances of instances in a particular availability zone.

Why Are You Monitoring?

There are a number of reasons for setting up a monitoring system for your application and infrastructure. The most obvious reason is that when things break, operators should be alerted by automated systems, rather than phone calls from upset customers or C-level executives.

Another reason is to allow systems administrators to identify trends in application usage so they can make informed decisions about capacity requirements. Knowing how your application performed last month is critical when it comes to planning your requirements for next month.

Yet another reason is to allow administrators and developers to accurately measure the effects of infrastructure changes and new application features. It is difficult to improve what you do not measure. If your application is running slowly, a well-planned monitoring system will allow you to quickly identify and remove bottlenecks. You need metrics demonstrating that your changes are having a positive effect in order to make continual, incremental improvements, to keep the aforementioned C-level executives happy. The opposite is also true—it is very easy to accidentally reduce the performance of your application when deploying new code on a frequent basis. In this case, it is imperative to recognize which changes are negatively affecting performance so that they can be reverted or fixed.

CloudWatch

Amazon's own CloudWatch service is the starting point for many administrators when it comes to monitoring AWS services. In fact, many AWS services, such as Auto Scaling, rely on CloudWatch to perform scaling operations, making it an essential part of the infrastructure. For example, CloudWatch is responsible for monitoring metrics such as the CPU loads of EC2 instances. When these metrics cross certain thresholds, the Auto Scaling system is alerted so that it can take the relevant action of spawning or terminating instances according to an Auto Scaling Policy.

CloudWatch is also an integral part of Amazon's Health Check feature, used by Elastic Load Balancers to identify instances that have failed or are otherwise "unhealthy".

In addition to tracking built-in metrics, such as disk usage and CPU load, CloudWatch can monitor custom metrics provided by external sources. If the built-in metrics do not provide the detail required to inform your Auto Scaling requirements, these custom metrics can provide more granular control when scaling Auto Scaling Groups up or down. The sky's the limit when using Custom Metrics. Typically, administrators monitor values such as requests per second, although more outlandish metrics such as solar flare activity or the phase of the moon could be used if it makes sense for your application.

Although powerful, CloudWatch is not perfect and has a number of drawbacks. The largest, in my opinion, is its web-based interface. When monitoring a large number of metrics, the interface can become slow and cumbersome, taking a long time to display the results you are looking for. This is because graphs are generated on demand each time they are viewed. So if you have many metrics, or are viewing data across a large time range, generating these graphs can take a long time and become quite frustrating.

Furthermore, a cost is associated with submitting Custom Metrics to CloudWatch. The cost is based on the number of metrics submitted, and the number of API requests required to submit these requests. At the time of writing, the cost is $0.50 per metric per month, plus $0.01 per 1000 API requests. In most cases, each metric submission will require one API PUT request. Assuming you are submitting your custom metric every minute, this will result in a cost of around $0.43 per month. While not overly expensive, these costs can quickly add up and must be taken into consideration when building your monitoring system. Of course, building a custom monitoring system will have other costs: the time taken to implement it, licensing costs for third-part monitoring services, and so on.

Auto Scaling and Custom Metrics

One of the most useful features of CloudWatch is its integration with Auto Scaling. This is commonly used to increase or decrease capacity in an Auto Scaling group according to metrics such as CPU utilization. When your instances are becoming too busy to cope with demand more instances are launched. As demand decreases the surplus instances are gradually terminated.

Auto Scaling is not limited to metrics that are built in to CloudWatch: It is also possible to scale up or down based on the values of custom metrics that you provide to CloudWatch. As an example, consider a task-processing application where tasks are queued in a messaging system and processed by EC2 instances. When running such an application, you might want the number of EC2 instances to scale dynamically according to the number of tasks waiting in the queue. If your message processing system is based on Amazon's Simple Queue Service, you are able to use CloudWatch's built-in metrics (such as the `ApproximateNumberOfMessagesVisible` SQS metric, which shows the number of messages available for retrieval in the queue) to control the size of your Auto Scaling group.

If you are using something other than SQS to store your queued messages for your task processing application, you will need to provide CloudWatch with the data yourself so that Auto Scaling can make decisions based on these metrics.

Custom metrics do not need to be pre-defined: simply send the metric to CloudWatch and it will begin storing and graphing it for you. This can be done in a number of ways. The Amazon API can be used from language-specific libraries (such as Boto for Python or Fog for Ruby), or by sending requests using the RESTful API. The most straightforward method is to use the AWS CLI tool.

In the following example, we will create a WaitingTasks custom metric, which performs the same function as the `ApproximateNumberOfMessagesVisible` metric for SQS-based systems. Once CloudFormation has some data on your custom metric it can be used in the same way as built-in metrics to control Auto Scaling processes.

To begin sending our custom metric to CloudWatch, execute the following command:

```
aws cloudwatch put-metric-data --namespace "MyAppMetrics" --metric-name WaitingTasks --value 20 --
```

This example creates a WaitingTasks metric and provides an initial value of 20. After a few moments this metric will become visible in the CloudWatch Web Console.

Instead of providing values on the command line, you can achieve the same result by sending a JSON file containing the metric data. For example, you could create a file named *metric_data.json* containing the following contents:

```
[
  {
    "MetricName": "WaitingTasks",
    "Value": 20,
    "Unit": "Count"
  }
]
```

This file can be uploaded with the following command:

```
aws cloudwatch put-metric-data --namespace "MyAppMetrics" --metric-data file://metric_data.json
```

This is equivalent to the first command in this section and is most useful when providing more complex or detailed metrics.

Chapter 6 describes how to create and manage Auto Scaling Groups. In this section we will create a CloudFormation stack that describes an Auto Scaling Group that dynamically shrinks and grows based on the value of our WaitingTasks metric. An example of such as a stack is:

```
{
    "AWSTemplateFormatVersion" : "2010-09-09",
    "Description" : "Auto Scaling on Custom Metrics",
    "Resources" : {
        "CustomMetricLaunchConfig" : {
          "Type" : "AWS::AutoScaling::LaunchConfiguration",
          "Properties" : {
            "ImageId" : "ami-12345678",
            "InstanceType" : "m1.small"
          }
        },
        "CustomMetricScalingGroup" : {
            "Type" : "AWS::AutoScaling::AutoScalingGroup",
            "Properties" : {
                "AvailabilityZones" : [ "eu-west-1" ],
                "Cooldown" : "300",
                "DesiredCapacity" : "1",
                "LaunchConfigurationName" : "CustomMetricLaunchConfig",
                "MaxSize" : "10",
                "MinSize" : "1"
            }
        },
```

```
"ScaleUpPolicy" : {
    "Type" : "AWS::AutoScaling::ScalingPolicy",
    "Properties" : {
        "AdjustmentType" : "ChangeInCapacity",
        "AutoScalingGroupName" : { "Ref" : "CustomMetricScalingGroup" },
        "ScalingAdjustment" : "1"
    }
},
"ScaleDownPolicy" : {
    "Type" : "AWS::AutoScaling::ScalingPolicy",
    "Properties" : {
        "AdjustmentType" : "ChangeInCapacity",
        "AutoScalingGroupName" : { "Ref" : "CustomMetricScalingGroup" },
        "ScalingAdjustment" : "-1"
    }
},
"WaitingTasksAlarm" : {
    "Type" : "AWS::CloudWatch::Alarm",
    "Properties" : {
        "AlarmActions" : [ { "Ref" : "ScaleUpPolicy" } ],
        "ComparisonOperator" : "GreaterThanThreshold",
        "EvaluationPeriods" : "1",
        "MetricName" : "WaitingTasks",
        "Namespace" : "MyAppMetrics",
        "OKActions" : [ { "Ref" : "ScaleDownPolicy" } ],
        "Statistic" : "Maximum",
        "Threshold" : "10",
        "Unit" : "Count"
    }
}
}
}
}
```

This stack consists of a number of components, which are all required to make everything work correctly.

The CustomMetricLaunchConfig and CustomMetricScalingGroup should be familiar from Chapter 6. These are a required part of any Auto Scaling Group.

Next we have the ScaleUpPolicy and ScaleDownPolicy. These Scaling Policy resources control *how* the capacity of an Auto Scaling Group should be changed. The scale-up policy has a ScalingAdjustment parameter of 1, which means a single additional EC2 instance should be launched in to the Auto Scaling Group when this policy is triggered. The scale-down policy's ScalingAdjustment parameter is -1, meaning a single instance should be removed from the group when this is triggered.

The ScalindAdjustment parameter controls what changes will be made to the size of the Auto Scaling Group when this policy is triggered. The AutoScalingGroupName parameter associates the Scaling Policy with a particular Auto Scaling Group.

The final component is the WaitingTasksAlarm, which ties everything together and controls *when* the capacity should be changed. The important parts of this resource are the AlarmActions and OKActions parameters, which state what should happen when it enters and leaves the Alarm state. It enters the Alarm state when the specified metric —in this case, the WaitingTasks metric in the MyAppMetrics namespace—is over the threshold of 10 for a single evaluation period. That is, as soon as CloudWatch notices this value is above 10, it will put this alarm in to the Alarm state, triggering the ScaleUpPolicy.

This will cause an additional instance to be launched in to the Auto Scaling Group. Once this instance begins processing tasks, the WaitingTasks value will drop below 10, which puts the WaitingTasksAlarm in the OK state. This causes the ScaleDownPolicy to be triggered, resulting in a single instance in the Scaling Group being terminated.

The CustomMetricScalingGroup has its Cooldown parameter set to 300. This value, measured in seconds, controls how frequently Auto Scaling events occur for this group. By setting it to five minutes (300 seconds) we ensure that there is a gap between instances being created and deleted.

To see this in action, create the CloudFormation stack using the template just shown. Because we previously set the value of the WaitingTasks metric to 20, a new instance will be launched. Wait for the two instances to finish launching, and then issue the following command:

```
aws cloudwatch put-metric-data --namespace "MyAppMetrics" --metric-name WaitingTasks --value 5 --u
```

This makes CloudWatch think there are only 5 messages remaining in the queue, which will trigger the ScaleDownPolicy. After a few moments one of the instances in the Scaling Group will be terminated.

In the real world we would be periodically executing a command that checks the size of the waiting tasks queue and submits the value to CloudWatch, making this an entirely automated process.

If no data is available for the requested metric, Auto Scaling will, by default, take no action. This behavior can be controlled by adding an InsufficientDataActions parameter to the WaitingTasksAlarm resource. For more information on controlling this behavior see the documentation for the AWS::CloudWatch::Alarm resource type.

Old Tools, New Tricks

Many common monitoring tools predate AWS significantly. Nagios, popular open-source monitoring tool, has been around since 1999, seven years before the introduction of EC2. One of the advantages of Nagios (and other mature monitoring tools) is the ecosystem surrounding it, providing features such as graphing, reporting, and integration with third-party services. Using such tools you can build a replacement for Cloud-

Watch that might be more suited for your specific needs, as well as work around some of the limitations mentioned in the previous section.

> The Nagios Exchange contains a number of plugins that can be used to integrate Nagios and AWS. In addition to plugins that directly monitor the status of your EC2 instances, there are plugins that query CloudWatch and other AWS components, allowing you to monitor your AWS infrastructure in a variety of ways. Remember that any tools that pull data from CloudWatch will do so by querying the CloudWatch API. Pulling these metrics too frequently will result in a higher AWS bill.

As Robert Heinlein pointed out in *The Moon is a Harsh Mistress*, there's no such thing as a free lunch. The time taken to implement a custom monitoring solution is time taken away from building your core application infrastructure, so this undertaking should be carefully considered against off-the-shelf tools, or indeed simply using CloudWatch and learning to love it despite its limitations.

At a high level, there are two main ways of dynamically configuring tools like Nagios within an AWS infrastructure.

The first is to use a configuration management tool such as Puppet. Puppet provides a feature known as Exported Resources, which allows the configuration of one node to influence the configuration of another node. For example, when Puppet runs on one of your web application instances, it can use the data collected on this node to dynamically configure your monitoring instances. If Puppet recognizes that Nginx is running on the web application node, the Nagios instance will be automatically configured to run Nginx-specific checks against the web application node, such as making sure HTTP requests to port 80 respond with an HTTP 200 status code.

This feature relies on PuppetDB, which means it will work only when Puppet is running in the traditional master/client mode.

Implementing this system requires two entries in your Puppet manifest files. The first is the declaration stage, where you declare the virtual resource type. This stanza is placed in the Puppet manifest for the host to be monitored. For example, to monitor Nginx as just described, the following virtual resource declaration can be used:

```
@@nagios_service { "check_http${hostname}":
    use                 => 'http-service',
    host_name           => "$fqdn",
    check_command       => 'check_http',
    service_description => "check_http${hostname}",
    target              => '/etc/nagios/conf.d/dynamic_${fqdn}.cfg',
    notify              => Service[$nagios::params::nagios_service],
}
```

Notice that the resource type (in this case, `nagios_service`) is prefixed with `@@`. This lets Puppet know that this is a virtual resource that will be realized on another node. Declaring this virtual resource will not cause any changes to be made on the monitored instance itself. Rather, it causes the relevant data to be stored in PuppetDB for later use by another node.

This declaration will configure Nagios to perform a simple HTTP service check, using the `check_http` command. The `target` parameter writes this configuration to a file whose name contains the fully-qualified domain name of the monitored instance. If your monitored instance has an FQDN of *web01.example.com*, this configuration segment would be written to a file named */etc/nagios/conf.d/dynamic_web01.example.com.cfg* on the Nagios host. By default Nagios will include all *.cfg* files contained within the */etc/nagios/conf.d* directory when the configuration is read. The `notify` parameter causes Nagios to be restarted whenever this file changes, so that the new monitoring configuration is picked up automatically.

The second component is the collection stage, which affects the Nagios instance. The following line should be added to the node definition for the Nagios instance in your Puppet manifest file:

```
Nagios_service <<| |>>
```

When Puppet is run on the Nagios instance, any previously-declared virtual resources describing Nagios resources will be realized on the Nagios instance. This is the point at which the */etc/nagios/conf.d/dynamic_web01.example.com.cfg* file is created and the Nagios service restarted.

Although support for Nagios is explicitly built in to Puppet's Exported Resources feature, there is nothing to stop it being used for other packages. In fact it can be used to configure any service that relies on text-based configuration files, making it a very flexible tool for dynamic configuration.

Another method of achieving this goal is to use a custom script to query the AWS API and write Nagios configuration files based on the retrieved data. This is a good option if you do not have an existing Puppet master/client infrastructure in place. Implementing Puppet merely to take advantage of Exported Resources could be considered overkill, making the custom script route a more attractive option.

One potential downside of this system relates to instances that get terminated as part of expected Auto Scaling operations. Nagios must be informed that the instance no longer exists and shouldn't be monitored. For this reason, I typically recommend putting a separate configuration file in the Nagios configuration directory for each node, hence the use of `dynamic_${fqdn).cfg` in the example.

Auto Scaling can be configured to send a notification to Amazon's Simple Notification Service when instances are launched or terminated. Subscribing to these notifications

makes it simple to delete all of the Nagios configuration files for a particular host once it is terminated. After Nagios is restarted, the host will no longer be monitored, and will not cause any false alarms once Nagios notices that the instance is no longer accessible to its checks.

Sensu and Other Monitoring Options

The above section described how to use Nagios to monitor EC2 instances. Of course, Nagios is only one of the wide range of tools that can be used for system monitoring, and I chose it because it still enjoys a high level of popularity among system administrators. This is, in part, due to the number of plugins available for Nagios allowing almost any service or application to be monitored.

There are many other highly-regarded packages that can be used to monitor your services. One example is Icinga, which is a fork of the open-source version of Nagios. Icinga aims to provide compatibility with all existing Nagios plugins while making enhancements to the web interface and core software.

Another package in this space is Sensu. Sensu has a number of features that make it an excellent choice for monitoring cloud-based infrastructure. Chief among these is the architecture of Sensu itself. Rather than operating in a client/server mode (as do Nagios and Icinga) it makes use of RabbitMQ, an AMQP-based messaging system. This makes it inherently more scalable than software that relies on a single central master server.

When a service is checked, the Sensu client writes the check result to a RabbitMQ queue, where it is read by the Sensu server process. Decoupling the submission of a check from the reading process in this way enables a much higher throughput than an architecture in which clients submit their check results directly to the master. Because RabbitMQ can operate in a highly-available cluster, it is also more resilient to failure. As long as both the master and client processes know the address of the RabbitMQ server, check results can be submitted. If the master server is briefly unavailable, messages will wait in the queue until the master is available again.

When combined with Auto Scaling Groups, this configuration of Sensu makes the entire monitoring system more reliable. Should the EC2 instance running the master server be terminated due to failure, a new one can be brought into service and it will begin processing all of the waiting check results automatically.

Backups

Introduction

If one relies solely on marketing materials provided by cloud hosts and resellers, one might be forgiven for thinking that the cloud is a magical place where nothing breaks and everything Just Works. Unfortunately, this is not the case. Cloud-based infrastructure requires just as much backup planning as a traditional self-hosted architecture—sometimes more, due to the dynamic nature of the cloud. Fortunately there is a silver lining: along with new challenges, the cloud provides new features to make backups simpler and reduce implementation time.

Although business types can think of "the cloud" as a single logical entity, we must look beyond that high-level presentation to view our cloud as a series of datacenters spread across multiple regions, and plan our backups accordingly. To run a highly available service, you would not put all of your servers in a single datacenter. You should plan backups with the same way consideration in mind.

Furthermore, it is also important to think about off-site backups. When working with AWS this can mean one of two things: storing backups outside of your primary region, or going a step further and storing them entirely outside of AWS.

There are two separate risks against which you are trying to protect yourself. If a single AWS region goes down, it would be relatively simple to deploy your operations to another region. For example, if you host your application in us-east-1 and store your backups in eu-west-1, you can redeploy your application to us-east-2, and restore servers from the backups in eu-west-1 to get your application up and running again.

However, if the AWS API is unavailable, it can become impossible to retrieve these backups no matter which region they are hosted in, rendering them useless in protecting against this particular failure mode.

Backups are a means to an end, not an end in themselves. What we are trying to achieve is a way to restore operations in the event of failure, no matter the cause or severity of this failure. Unless your backups put you in a position where you can restore after failures they are of no use whatsoever.

An untested backup procedure is useless. In fact, an untested backup procedure can be worse than no backups at all, as it provides a false sense of security. Perform regular restore tests to make sure that your documented backup procedure works as planned.

This chapter looks at some of the ways traditional tools and AWS-specific features can be used to implement reliable backup procedures.

RDS Database Snapshot

If you are using Amazon's RDS service to host your database, you can make use of the RDS Database Snapshot feature. This process can be automated so that Amazon automatically backs up your database according to your specified schedule, or you can manually create backup snapshots before performing potentially destructive operations.

When you use RDS, automated snapshots are automatically enabled and will be carried out during the maintenance window chosen when creating the database instance. The process of enabling and disabling automated backups is described in Amazon's Working With Automated Backups documentation. You can find a more general explanation of how automated backups work in DB Instance Backups. The Related Topics section of the latter page provides more detail on working with automated backups.

If you rely on RDS snapshots, it is important to keep track of the most recent snapshot ID when backing up the other files required for your application to ensure that the database schema referenced in your code matches the data contained in the snapshot. This can be done by regularly querying the ID of the most recent snapshot and storing it in a text file alongside the other application files and making sure it is included in the backup. When restoring from backups, this file will let you know which corresponding DB snapshot should be restored.

Finally, even if you are using RDS, you might wish to follow the other steps in this chapter to regularly take a full dump of the database. This will ensure that you can restore the backup to a non-RDS database. RDS snapshots can only be used to restore to RDS databases, and do not provide any facility to make off-site backups.

At the time of writing, RDS supports MySQL, Oracle, Microsoft's SQL Server, and PostgreSQL, all of which can be used with automated backups. Because PostgreSQL is a relatively recent addition to the list of supported database engines, a lot of people are still running their own PostgreSQL database clusters on EC2 instances, outside of RDS. Later in this chapter we will look at ways to manually backup databases that are not part of RDS.

Backing up Static Files From EC2 Instances to S3

One common use of EC2 is to host web applications such as WordPress blogs. This section will describe how an EC2-hosted WordPress blog can be backed up, with the backup archive being stored on S3. Although the actual steps taken will differ if you are using a different application—or indeed, your own custom application—the general steps will be the same.

Dynamic websites usually consist of two major components that need to be backed up: the database in which content and configuration options are stored, and static files such as images and HTML files. Furthermore, these websites might allow users to upload their own files, such as profile images for blog authors. All of these components need to be backed up to ensure the entire site can be restored in the event of an EC2 instance failing.

If you are backing up a custom application, I'll assume that the actual code that powers your application is stored in an external source control system such as GitHub. Therefore, backing up these files is outside of the scope of this section. If you are not using version control for your application, you could back up the code files as part of the file-based backup archive.

The first step in the backup process is to create a snapshot of your database contents.

For the sake of simplicity, we will assume the WordPress blog is configured to use a MySQL instance running on the same host as WordPress. If your database is hosted on a separate instance, you will need to modify some of the example commands to connect to the correct host in order to dump the contents of the database.

The first step is to create a database dump file containing the SQL statements required to recreate your database. MySQL helps you do this with the conveniently named *mysqldump* command. For example, if your WordPress database is named *my_blog*, the following command can be used to dump its contents to a file located at */var/backups/ my_blog/database.sql*:

```
mysqldump my_blog > /var/backups/my_blog/database.sql
```

After the database dump has completed, you can move on to backing up the application files. Let's assume they are stored in */srv/vhosts/my_blog*. First copy all of the files in to the backups directory with the following command:

```
rsync -av /srv/vhosts/my_blog/ /var/backups/my_blog/
```

This command will copy all files. The `-a` option indicates that rsync should run in archive mode, which, among other things, ensures that file permissions and ownerships are copied.

To reduce the amount of data stored in S3, you can create a compressed tar archive before transferring it to S3. This is done with:

```
DATE=`date -u +"%Y%m%d%H%M"`
BACKUP_FILE="/var/backups/my_blog_${DATE}.tgz"
tar zcvf ${BACKUP_FILE} /var/backups/my_blog/
```

Finally, you can transfer the resulting file to S3 with the *s3cmd* command:

```
s3cmd put "${BACKUP_FILE}" s3://my-blog-backups/
```

To keep the WordPress instance clean, delete the temporary files used during the backup:

```
rm -rf /var/backups/my_blog/*
```

This last step is optional. Leaving these files in place would speed up subsequent backups, because *rsync* would have a smaller amount of data to transfer, needing to transfer only files that had changed since the last backup.

Put these commands together to make a script that can be easily used to automatically back up the database and files required to recover your blog in the event of a disaster.

Restoring the resulting archive is simply a case of extracting the files to the correct location and importing the SQL statements file into the database. For example:

```
cd /srv/vhosts/myblog
tar xcvf my_blog_201206011200.tar.gz

mysql my_blog < myblog/database.sql
rm myblog/database.sql
```

Later in this section we will look at ways to move this data out of S3 and in to off-site storage for additional reliability.

Rolling Backups with S3 and Glacier

When keeping backups on S3, it is important to keep track of how much data you are storing. Each additional byte will gradually increase your monthly Amazon bill, so there is no point storing data that you will never need to use. Your backup strategy should reflect this. In many ways, the traditional approaches used with tape backups are still applicable: one tape per day of the week, one tape for each of the last four weeks, and finally one tape for each month. A total of twenty-three tapes would allow you to restore from any day of the previous week, any week of the previous month, and any month of the previous year.

A similar approach can also be used in S3 to keep your data storage requirements to a minimum. In this section we will look at how this can be implemented using S3's Object LifeCycle Management and Object Archival features. This method relies on S3 for the last month's worth of backups, and monthly backups are automatically transitioned to the Glacier archival service on the assumption that they will be required less frequently than daily or monthly backups.

Let's assume that you are using one of the methods described in this chapter to store your backups in S3. We will configure S3 to automatically transition backup objects to Glacier once they reach a certain age, and delete them automatically from S3. This will keep your S3 backups bucket clean and ensure that you do not gradually build up a huge Amazon bill for storing unecessary files.

For this example we will assume that your daily backup archives are prefixed with "/backups/daily/" in your S3 bucket, for example */backups/daily/20131003.tar.gz*.

The rules that govern when S3 objects are transitioned to Glacier are stored as a *lifecycle* subresource on your S3 bucket. A lifecycle configuration can contain up to 1000 rules controlling when your files—or subsets thereof—are transitioned to Glacier. Lifecycle configurations can be created and modified using the AWS API, or via the AWS Web Console. In this example we will use the Web Console to create our lifecycle configuration.

Begin by finding your S3 bucket in the AWS Web Console. Right-click on the bucket object and select Properties, then navigate to the Lifecycle tab shown in Figure 12-1.
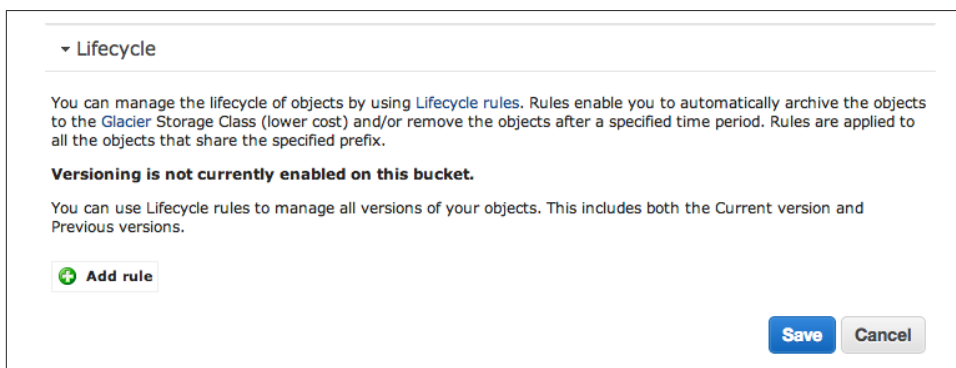

Figure 12-1. Lifecycle Properties

Click Add Rule to open the Lifecycle Rule wizard shown in Figure 12-2.
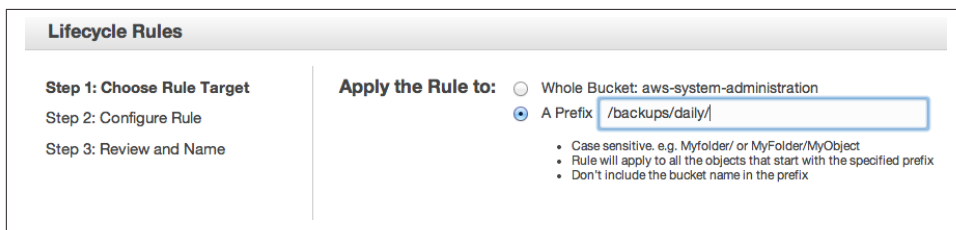

Figure 12-2. Choose Rule Target

The first screen lets us choose which objects the chosen action will be applied to. In the Prefix box, enter "/backups/daily/", as we want this particular rule to apply only to the daily backup archives. Click on Configure Rule to proceed to the next screen shown in Figure 12-3. This screen lets us choose what action will be performed on the objects. Select `Archive and then Permanently Delete` to see the options relevant to the rule we are creating.

Figure 12-3. Configure Rule

Our desired rule consists of two actions: transitioning the object to Glacier, and removing the original S3 object once it has been transitioned to Glacier. Because this is such a common pattern Amazon allows us to configure both of these actions from a single screen.

Enter 31 in to the `Archive to the Glacier Storage Class` input box, and 32 in the `Permanently Delete` input box as shown in the image above.

In this example, both of these time values should be set to 31 days. Lifecycle rules can be set only in daily increments so we set this to 31 to ensure we do not transition objects too early in months with fewer than 31 days. The object can not be deleted before it is archived, so we must wait a day in between archiving the objects to Glacier, and deleting the original objects from the S3 bucket.

Click `Review` to confirm the options you have selected. On this screen you can provide the rule with an optional name. Finally, click `Create and Activate Rule`.

You can now store your backups in S3 and all objects will be moved to Glacier once they are older than 31 days.

At the time of writing, there is no equivalent for deleting or moving objects out of Glacier, which means you will still need to manually remove objects from your Glacier account once you are sure they are no longer required. "Manually" in this case might mean deleting the objects via the Glacier Web Console or writing a script that searches for archives older than (for example) six months and deleting them.

PostgreSQL and Other Databases

Amazon offers hosted and managed PostgreSQL databases via its RDS service. For a long time RDS supported only MySQL databases, so it was common for PostgreSQL users to run their own PostgreSQL instances on EC2 instances. This also meant that users were responsible for providing their own backup procedures instead of relying on the backup features provided by RDS. For this reason, this section will go in to some detail about the manual process of backing up databases that are hosted on EC2 instances, as opposed to running on RDS. While this section uses PostgreSQL-specific commands, the general principles are applicable to most database engines.

Backing up dynamic databases is not as simple as copying the database files to a remote location. First, PostgreSQL must be informed that a database backup is about to executed. If you forget this step, the database files could be copied while they are effectively in an unusable state, making restoration procedures either very time-consuming or in some cases impossible. PostgreSQL must be informed that a database backup is about to be executed so that it can flush any in-memory data to disk and ensure the files on disk are in a state that will allow them to be used to restore data.

There are two main methods of backing up a PostgreSQL database running on EC2: *pg_dump* and snapshots. The latter is more complicated, but better for large database.

pg_dump

pg_dump and its companion commands are distributed with PostgreSQL. *pg_dump* dumps all of the data from the specified database, without interfering with reads and writes. Any changes made to the database after you start *pg_dump* will not appear in the dump, but the dump will be consistent. Data can be dumped in a variety of formats, which can be restored with the *pg_restore* command. The default option is to create a file containing SQL statements, but it is also possible to dump data in a PostgreSQL-specific format or create a tar archive of the data.

This method is especially suited for smaller databases (less than 5GB) because the resulting dump file can be transferred directly to S3. For example, to dump the *my_db* database and store it in an S3 bucket named `my-db-backups`, you could use the following script:

```
#!/bin/bash -e

DB_NAME="my_db"
BUCKET_NAME="my-db-backups"
DATE=`date -u +"%Y%m%d%H%M"`
BACKUP_DIR="/var/backups"
BACKUP_FILE="${BACKUP_DIR}/${DB_NAME}_${DATE}.tar"
mkdir -p $BACKUP_DIR

# Dump the database as a tar archive
```

```
pg_dump ${DB_NAME} --file="${BACKUP_FILE}" --format=tar

# Copy the tar file to S3
s3cmd put "${BACKUP_FILE}" s3://${BUCKET_NAME}

# Delete the local tar file
rm ${BACKUP_FILE}
```

This script first dumps the data from PostgreSQL using the *pg_dump* command. Once the backup file has been created, it is copied to an S3 bucket. Finally, the local copy of the tar file is deleted to ensure you do not gradually use up all of the space available on the device on which */var* is mounted. Bash's `-e` option is used is used to ensure that the script fails immediately if any of the commands in the script fails.

To restore a database backed up using this script, simply copy the backup file from the S3 bucket onto the new PostgreSQL instance and use the *pg_restore* command to load the data in to the new database. For example:

```
pg_restore --dbname=my_db /path/to/backup/file.tar
```

As your database grows, the time taken to run the *pg_dump* and *pg_restore* commands will increase, making this option less attractive. If you wish to make backups once per hour, this process will become useless as soon as it takes longer than sixty minutes to complete, because you will never be able to complete a backup before a new one is started.

Snapshots and Continuous Archiving

Another option for backing up EC2-hosted PostgreSQL databases is to make use of the snapshotting capabilities of EBS volumes. This is slightly more complex, but provides a much quicker way of backing up larger databases. This method does not produce a single file that can be used with the *pg_restore* command. Instead, it uses PostgreSQL's base backup feature to produce one or more EBS snapshots that can be used to provision new EBS volumes containing your backed up data.

This method relies on PostgreSQL's Continuous Archiving features. Configuring this is beyond the scope of this book. Refer to the PostgreSQL documentation on Continuous Archiving for information on how to configure and test this feature.

In a nutshell, Continuous Archiving will periodically store your data in an external location so that it can be used for a restore later. This is done by archiving the "write-ahead log" (WAL) files, which essentially "play back" all operations performed on your database. This allows you to restore the database to a time of your choosing ("point-in-time recovery"). However, playing back all of the WAL files can take some time. To reduce restoration time, create a "base backup" that is used as a starting point for re-covery. This allows you to restore a smaller number of WAL files in external storage, and play back only WAL files that were created after a particular base backup was taken.

WAL-E is a program designed to help creating and managing Post-greSQL WAL files and the process of creating base backups. Although it is still worth learning and understanding how the underlying concepts of WAL and base backups work, WAL-E can make the day-to-day usage of Continuous Archiving a lot simpler and more reliable. It can be found at *https://github.com/wal-e/wal-e*.

For performance reasons it is recommended that you put PostgreSQL's data directory on its own EBS volume. The data directory location will vary depending on the version of PostgreSQL and the operating system in use. For example, a PostgreSQL 9.4 instance on Ubuntu will be stored in */var/lib/postgresql/9.3/main*. Attaching an additional EBS volume to your instance and mounting the data directory on this volume will improve performance, because PostgreSQL will not be contending with the operating system in order to read and write files.

PostgreSQL's tablespace feature allows you to store each table in a separate on-disk location. This feature makes it possible to store each table on a different EBS volume, further improving performance. In conjunction with EBS Provisioned IOPS, which provide a higher level of performance than vanilla EBS volumes, this can dramatically improve the performance of disk-bound workloads.

Backing up

This section, and the following, assume that you have at least two EBS volumes attached to your PostgreSQL EC2 instance. The first volume is used to mount the root of the file system (the **/** directory). The second volume is used solely for PostgreSQL's data directory and is mounted at */var/lib/postgresql/9.3/main*. For example purposes, the data EBS volume will be created with the device name */dev/sda2*, a typical name for a Linux device responding to the popular SCSI protocol. It is further assumed that you have installed PostgreSQL and created a database for testing purposes according to PostgreSQL's documentation.

Begin by connecting to the PostgreSQL instance as the superuser by running the *psql* command. Depending on how you configured your database, you might need to provide the superuser password at this point. Once connected to the database, issue the *pg_start_backup* command:

```
SELECT pg_start_backup('test_backup');
```

This command will create a "backup label" file in the data directory containing information about the backup, such as the start time and label string. It is important to retain this file with the backed-up files, because it is used during restoration. It will also inform PostgreSQL that a backup is about to be performed so that a new checkpoint can be

created and any pending data can be written to disk. This can take some time, depending on the configuration of the database.

Once the command completes, make a snapshot of the EBS volume on which the PostgreSQL database is stored.

> The filesystem type you are using might require a slightly different approach at this stage. This example assumes you are using XFS, which requires that the file system be frozen before a snapshot is made. This ensures that any attempts to modify files on this filesystem will be blocked until the snapshot is completed, ensuring the integrity of the files.

First, freeze the file system with the xfs_freeze command:

```
xfs_freeze -f /var/lib/postgresql/9.3/main
```

Once the filesystem has been frozen, it is safe to take a snapshot using the EBS snapshotting tools. This can be done from the AWS Web Console or from the command-line tools. Because the final goal is to use this process as part of an automated backup script we will use the command line tools.

```
# Find out the ID of the instance on which this command is executed
EC2_INSTANCE_ID=$(ec2metadata --instance-id)
# The device name is used to filter the list of volumes attached to the instance
DEVICE="/dev/sda2"
# Find the ID of the EBS volume on which PostgreSQL data is stored
VOLUME_ID=(`ec2-describe-volumes --filter="attachment.instance-id=${EC2_INSTANCE_ID}" --filter="at

ec2-create-snapshot ${VOLUME_ID}
```

These commands will query the EC2 API to find out the volume ID of the EBS volume containing the PostgreSQL data before using the *ec2-create-snapshot* command to create a snapshot of this volume. The ID of the new snapshot will be printed as part of the output from *ec2-create-snapshot*.

Although the snapshot has not yet been fully created, it is safe to begin using the volume again. AWS will create a snapshot of the volume in the state it was in when the *ec2-create-snapshot* command was executed, even if data is subsequently changed.

As soon as the snapshotting command finishes, the XFS volume can be unfrozen with:

```
xfs_freeze -u /var/lib/postgresql/9.3/main
```

Finally, you can inform PostgreSQL that the backup has been completed and that is should resume normal operations:

```
SELECT pg_stop_backup();
```

Note that the backup is not entirely complete until the WAL archives identified by the *pg_stop_backup* command have been archived (by way of the `archive_command` configuration directive in postgresql.conf).

Restoring

Restoring from a backup created with this method is much simpler, if more time-consuming. Whether your original PostgreSQL instance has suffered from some kind of catastrophic failure or you are restoring the data to a new machine (perhaps for development or testing purposes) the procedure is essentially the same.

First you will need to launch a new instance using the same EBS configuration. That is, the new instance should use the same EBS volume layout as the original instance. There is one difference however: the data volume (*/dev/sda2*) should use the most recent snapshot created using the method described in the previous section. When the instance is launched, the snapshot will be used to create the sda2 device, ensuring that the PostgreSQL data is already in place when the instance is launched.

Once the instance is ready, you can log in and start PostgreSQL with the command:

```
/etc/init.d/postgresql-9.3 start
```

PostgreSQL must be configured using the same configuration files used for the original instance. The Continuous Archiving configuration directives will let PostgreSQL know how to restore the archived WAL files. This is controlled through the `restore_command` configuration directive.

During the startup process, PostgreSQL will recognize that the files in the data directory were created as part of a base backup and will automatically restore the required WAL files by executing the restore command. This can take some time depending on how much data was changed since the base backup was taken, and how quickly the archived WAL files can be restored from remote storage.

PostgreSQL will begin accepting connections as soon as it has finished restoring the WAL files and the database is up to date. At this point, the state of the data should match the state it was in just before the original database was terminated. If the original database was uncleanly terminated (for example, the actual EC2 instance was terminated) some data might be missing if PostgreSQL did not have time to archive the final WAL file(s).

As mentioned, this approach is a tad more complex than simply dumping a file full of SQL statements representing your data. However, it does provide a lot more flexibility in restoring your data, and is the only valid option once your database grows to the point where dumping SQL statements is no longer viable.

Off-site Backups

As I explained earlier, you may want to move some backups outside of the AWS eco-system (EC2 and S3) altogether for those rare but not impossible times that AWS goes down and the AWS API is unavailable. You can use another cloud provider to host your off-site backups, or provision a new server (entirely outside of AWS) to act as a backup host.

Assuming that your data has been backed up to S3 using one of the other methods described in this chapter, you can use tools such as *s3cmd* or *s3funnel* to regularly pull this data from S3 and store it on another server.

> Remember, backups without a reliable and regularly tested restore procedure can be worse than useless, providing nothing but a false sense of security.
>
> Test your restore procedures regularly!