Amazon Web Services IN ACTION

SECOND EDITION

Michael Wittig
Andreas Wittig

Foreword by Ben Whaley

MANNING

Compute & Networking

Abbr.	Name	Description	Where
EC2	Amazon Elastic Compute Cloud	Virtual machines with Linux and Windows	3
	AWS Lambda	Run code without the need for virtual machines	7
EIP	Elastic IP Address	Fixed public IP address for EC2 instances	3.6
ENI	Amazon EC2 Elastic Network Interface	Virtual network interface for EC2 instances	3.7
VPC	Amazon Virtual Private Cloud	Private network inside the cloud	6.5
	Amazon EC2 Security Group	Network firewall	6.4

Deployment & Management

Abbr.	Name	Description	Where
	AWS Elastic Beanstalk	Deployment tool for simple applications	5.4
	AWS OpsWorks	Deployment tool for multilayer applications	5.5
	AWS CloudFormation	Infrastructure automation and deployment tool	5.3
IAM	AWS Identity and Access Management	Secure access to your cloud resources (authentication and authorization)	6.3
CLI	AWS command-line interface	AWS in your terminal	4.2
SDK	AWS software development kits	AWS in your applications	4.3

Praise for the First Edition

Fantastic introduction to cloud basics with excellent real-world examples.

—Rambabu Posa, GL Assessment

A very thorough and practical guide to everything AWS … highly recommended.

—Scott M. King, Amazon

Cuts through the vast expanse of official documentation and gives you what you need to make AWS work now!

—Carm Vecchio, Computer Science Corporation (CSC)

The right book to program AWS from scratch.

—Javier Muñoz Mellid, Senior Computer Engineer, Igalia

Amazon Web Services in Action, Second Edition

MICHAEL WITTIG
ANDREAS WITTIG
FOREWORD BY BEN WHALEY

MANNING
Shelter Island

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Frances Lefkowitz
Technical development editor: John Hyaduck
Review editor: Aleksandar Dragosavljević
Project editor: Deirdre Hiam
Copy editor: Benjamin Berg
Proofreader: Elizabeth Martin
Technical proofreader: David Fombella Pombal
Typesetter: Gordan Salinovic
Cover designer: Marija Tudor

ISBN 9781617295119
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – DP – 23 22 21 20 19 18

brief contents

contents

foreword

Throughout the late 1990s and early 2000s I worked in the rank and file of system administrators endeavoring to keep network services online, secure, and available to users. At the time, administration was a tedious, onerous affair involving cable slinging, server racking, installing from optical media, and configuring software manually. It was thankless work, often an exercise in frustration, requiring patience, persistence, and plenty of caffeine. To participate in the emerging online marketplace, businesses of the era bore the burden of managing this physical infrastructure, accepting the associated capital and operating costs and hoping for enough success to justify those expenses.

When Amazon Web Services emerged in 2006, it signaled a shift in the industry. Management of compute and storage resources was dramatically simplified, and the cost of building and launching applications plummeted. Suddenly anyone with a good idea and the ability to execute could build a global business on world-class infrastructure at a starting cost of just a few cents an hour. The AWS value proposition was immediately apparent, ushering in a wave of new startups, data center migrations, and third-party service providers. In terms of cumulative disruption of an established market, a few technologies stand above all others, and AWS is among them.

Today, the march of progress continues unabated. In December 2017 at its annual re:Invent conference in Las Vegas, Werner Vogels, CTO of Amazon, announced to more than 40,000 attendees that the company had released 3,951 new features and services since the first conference in 2012. AWS has an $18 billion annual run rate and 40% year-over-year growth. Enterprises, startups, and governments alike have adopted the AWS cloud en masse. The numbers are staggering, and AWS shows no signs of slowing down.

Needless to say, this growth and innovation comes at the expense of considerable complexity. The AWS cloud is composed of scores of services and thousands of features, enabling powerful new applications and highly efficient designs. But it is accompanied by a brand-new lexicon with distinct architectural and technical best practices. The platform can bewilder the neophyte. How does one know where to begin?

Amazon Web Services in Action, Second Edition, slices through the complexity of AWS using examples and visuals to cement knowledge in the minds of readers. Andreas and Michael focus on the most prominent services and features that users are most likely to need. Code snippets are sprinkled throughout each chapter, reinforcing the programmable nature of the cloud. And because many readers will be footing the bill from AWS personally, any examples that incur charges are called out explicitly throughout the text.

As a consultant, author, and at heart an engineer, I celebrate all efforts to introduce the bewildering world of cloud computing to new users. *Amazon Web Services in Action, Second Edition* is at the head of the pack as a confident, practical guide through the maze of the industry's leading cloud platform.

With this book as your sidekick, what will you build on the AWS cloud?

—Ben Whaley, AWS community hero and author

preface

When we started our career as software developers in 2008, we didn't care about operations. We wrote code, and someone else was responsible for deployment and operations. There was a huge gap between software development and IT operations. On top of that, releasing new features was a huge risk because it was impossible to test all the changes to software and infrastructure manually. Every six months, when new features needed to be deployed, we experienced a nightmare.

Time passed, and in 2012 we became responsible for a product: an online banking platform. Our goal was to iterate quickly and to be able to release new features to the product every week. Our software was responsible for managing money, so the quality and security of the software and infrastructure was as important as the ability to innovate. But the inflexible on-premises infrastructure and the outdated process of deploying software made that goal impossible to reach. We started to look for a better way.

Our search led us to Amazon Web Services, which offered us a flexible and reliable way to build and operate our applications. The possibility of automating every part of our infrastructure was fascinating. Step by step, we dove into the different AWS services, from virtual machines to distributed message queues. Being able to outsource tasks like operating an SQL database or a load balancer saved us a lot of time. We invested this time in automating testing and operations for our entire infrastructure.

Technical aspects weren't the only things that changed during this transformation to the cloud. After a while the software architecture changed from a monolithic application to microservices, and the separation between software development and operations disappeared. Instead we built our organization around the core principle of DevOps: you build it, you run it.

We have worked as independent consultants since 2015, helping our clients get the most out of AWS. We've accompanied startups, mid-sized companies, and enterprises on their journey to the cloud. Besides designing and implementing cloud architectures based on AWS services, we are focusing on infrastructure as code, continuous deployment, Docker, serverless, security, and monitoring.

We enjoyed writing the first edition of our book in 2015. The astonishing support from Manning and our MEAP readers allowed us to finish the whole book in only nine months. Above all, it was a pleasure to observe you—our readers—using our book to get started with AWS or deepen your knowledge.

AWS is innovating and constantly releases new features or whole new services. Therefore, it was about time to update our book in 2017. We started to work on the second edition of our book in June. Within six months we updated all chapters, added three more chapters, and improved the book based on the feedback of our readers and our editors.

We hope you enjoy the second edition of *Amazon Web Services in Action* as much as we do!

acknowledgments

Writing a book is time-consuming. We invested our time, and other people did as well. We think that time is the most valuable resource on Earth, and we want to honor every minute spent by the people who helped us with this book.

To all the readers who bought the first edition of our book: thanks so much for your trust and support. Watching you reading our book and working through the examples boosted our motivation. Also, we learned quite a bit from your feedback.

Next, we want to thank all the readers who bought the MEAP edition of this book. Thanks for overlooking the rough edges and focusing on learning about AWS instead. Your feedback helped us to polish the version of the book that you are now reading.

Thank you to all the people who posted comments in the Book Forum and who provided excellent feedback that improved the book.

In addition, thanks to all the reviewers of the second and first edition who provided detailed comments from the first to the last page. The reviewers for this second edition are Antonio Pessolano, Ariel Gamino, Christian Bridge-Harrington, Christof Marte, Eric Hammond, Gary Hubbart, Hazem Farahat, Jean-Pol Landrain, Jim Amrhein, John Guthrie, Jose San Leandro, Lynn Langit, Maciej Drozdzowski, Manoj Agarwal, Peeyush Maharshi, Philip Patterson, Ryan Burrows, Shaun Hickson, Terry Rickman, and Thorsten Höger. Your feedback helped shape this book—we hope you like it as much as we do.

Special thanks to Michael Labib for his input and feedback on chapter 12 covering AWS ElastiCache.

Furthermore, we want to thank John Hyaduck, our technical developmental editor. Your unbiased and technical view on Amazon Web Services and our book helped to perfect the second edition. Thanks to Jonathan Thoms, the technical editor of the first edition as well.

David Fombella Pombal and Doug Warren made sure all the examples within our book are working as expected. Thanks for proofing the technical parts of our book.

We also want to thank Manning Publications for placing their trust in us. Especially, we want to thank the following staff at Manning for their excellent work:

- Frances Lefkowitz, our development editor, who guided us through the process of writing the second edition. Her writing and teaching expertise is noticeable in every part of our book. Thanks for your support.
- Dan Maharry, our development editor while writing the first edition. Thanks for taking us by the hand from writing the first pages to finishing our first book.
- Aleksandar Dragosavljević, who organized the reviews of our book. Thanks for making sure we got valuable feedback from our readers.
- Benjamin Berg and Tiffany Taylor, who perfected our English. We know you had a hard time with us, but our mother tongue is German, and we thank you for your efforts.
- Candace Gillhoolley, Ana Romac, and Christopher Kaufmann, who helped us to promote this book.
- Janet Vail, Deirdre Hiam, Elizabeth Martin, Mary Piergies, Gordan Salinovnic, David Novak, Barbara Mirecki, Marija Tudor, and all the others who worked behind the scenes and who took our rough draft and turned it into a real book.

Many thanks to Ben Whaley for contributing the foreword to our book.

Last but not least, we want to thank the significant people in our lives who supported us as we worked on the book. Andreas wants to thank his wife Simone, and Michael wants to thank his partner Kathrin, for their patience and encouragement.

about this book

Our book guides you from creating an AWS account to building fault-tolerant and auto-scaling applications. You will learn about services offering compute, network, and storage capacity. We get you started with everything you need to run web applications on AWS: load balancers, virtual machines, file storage, database systems, and in-memory caches.

The first part of the book introduces the principles of Amazon Web Services and gives you a first impression of the possibilities in the cloud. Next, you will learn about fundamental compute and network services. Afterward, we demonstrate six different ways to store your data. The last part of our book focuses on highly available or even fault-tolerant architectures that allow you to scale your infrastructure dynamically as well.

Amazon offers a wide variety of services. Unfortunately, the number of pages within a book is limited. Therefore, we had to skip topics such as containers, big data, and machine learning. We cover the basic or most important services, though.

Automation sneaks in throughout the book, so by the end you'll be comfortable with using AWS CloudFormation, an infrastructure-as-code tool that allows you to manage your cloud infrastructure in an automated way; this will be one of the most important things you will learn from our book.

Most of our examples use popular web applications to demonstrate important points. We use tools offered by AWS instead of third-party tools whenever possible, as we appreciate the quality and support offered by AWS. Our book focuses on the different aspects of security in the cloud, for example by following the "least privilege" principle when accessing cloud resources.

We focus on Linux as the operating system for virtual machines in the book. Our examples are based on open source software.

Amazon operates data centers in geographic regions around the world. To simplify the examples we are using the region US East (N. Virginia) within our book. You will also learn how to switch to another region to exemplarily make use of resources in Asia Pacific (Sydney).

Roadmap

Chapter 1 introduces cloud computing and Amazon Web Services. You'll learn about key concepts and basics, and you'll create and set up your AWS account.

Chapter 2 brings Amazon Web Services into action. You'll spin up and dive into a complex cloud infrastructure with ease.

Chapter 3 is about working with a virtual machine. You'll learn about the key concepts of the Elastic Compute Service (EC2) with the help of a handful of practical examples.

Chapter 4 presents different approaches for automating your infrastructure: the AWS command-line interface (CLI) from your terminal, the AWS SDKs to program in your favorite language, as well as AWS CloudFormation, an infrastructure-as-code tool.

Chapter 5 introduces three different ways to deploy software to AWS. You'll use each of the tools to deploy an application to AWS in an automated fashion.

Chapter 6 is about security. You'll learn how to secure your networking infrastructure with private networks and firewalls. You'll also learn how to protect your AWS account and your cloud resources.

Chapter 7 is about automating operational tasks with AWS Lambda. You will learn how to execute small code snippets in the cloud without the need of launching a virtual machine.

Chapter 8 introduces Amazon Simple Storage Service (S3), a service offering object storage, and Amazon Glacier, a service offering long-term storage. You'll learn how to integrate object storage into your applications to implement a stateless server by creating an image gallery.

Chapter 9 is about storing data from your virtual machines on hard drives with Amazon Elastic Block Storage (EBS) and instance storage. In order to get an idea of the different options available, you will take some performance measurements.

Chapter 10 explains how to use a networking filesystem to share data between multiple machines. Therefore, we introduce the Amazon Elastic File System (EFS).

Chapter 11 introduces Amazon Relational Database Service (RDS), which offers managed relational database systems like MySQL, PostgreSQL, Oracle, and Microsoft SQL Server. You will learn how to connect an application to an RDS database instance, for example.

Chapter 12 is about adding a cache to your infrastructure to speed up your application and save costs due to minimizing load on the database layer. Specifically, you will learn about Amazon ElastiCache, which provides Redis or memcached as a service.

Chapter 13 introduces Amazon DynamoDB, a NoSQL database offered by AWS. DynamoDB is typically not compatible with legacy applications. You need to rework your applications to be able to make use of DynamoDB instead. You'll implement a to-do application in this chapter.

Chapter 14 explains what is needed to make your infrastructure highly available. You will learn how to recover from a failed virtual machine or even a whole datacenter automatically.

Chapter 15 introduces the concept of decoupling your system to increase reliability. You'll learn how to use synchronous decoupling with the help of Elastic Load Balancing (ELB). Asynchronous decoupling is also part of this chapter; we explain how to use the Amazon Simple Queue Service (SQS), a distributed queuing service, to build a fault-tolerant system.

Chapter 16 dives into building fault-tolerant applications based on the concepts explained in chapter 14 and 15. You will create a fault-tolerant image processing web services within this chapter.

Chapter 17 is all about flexibility. You'll learn how to scale the capacity of your infrastructure based on a schedule or based on the current load of your system.

Code conventions and downloads

You'll find four types of code listings in this book: Bash, YAML, Python, and Node.js/JavaScript. We use Bash to create tiny scripts to interact with AWS in an automated way. YAML is used to describe infrastructure in a way that AWS CloudFormation can understand. In addition, we use Python to manage our cloud infrastructure. Also, we use the Node.js platform to create small applications in JavaScript to build cloud-native applications.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts. Sometimes we needed to break a line into two or more to fit on the page. In our Bash code we used the continuation backslash. In our YAML, Python, and Node.js/JavaScript code, an artificial line break is indicated by this symbol: ➥.

The code for the examples in this book is available for download from the publisher's website at https://www.manning.com/books/amazon-web-services-in-action-second-edition and from GitHub at https://github.com/awsinAction/code2.

Book forum

Purchase of *Amazon Web Services in Action, Second Edition* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://forums.manning.com/forums/amazon-web-services-in-action-second-edition. You can also learn more about Manning's forums and the rules of conduct at https://forums.manning.com/forums/about.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the authors

Andreas Wittig and Michael Wittig are software and DevOps engineers focusing on Amazon Web Services. The brothers started building on AWS in 2013 when migrating the IT infrastructure of a German bank to AWS—the first bank in Germany to do so. Since 2015, Andreas and Michael have worked as consultants helping their clients to migrate and run their workloads on AWS. They focus on infrastructure-as-code, continuous deployment, serverless, Docker, and security. Andreas and Michael build SaaS products on top of the Amazon's cloud as well. Both are certified as AWS Certified Solutions Architect - Professional and AWS Certified DevOps Engineer - Professional. In addition, Andreas and Michael love sharing their knowledge and teaching how to use Amazon Web Services through this book, their blog (cloudonaut.io), as well as online- and on-site trainings (such as *AWS in Motion* [https://www.manning.com/livevideo/aws-in-motion]).

about the cover illustration

The figure on the cover of *Amazon Web Services in Action, Second Edition* is captioned "Paysan du Canton de Lucerne," or a peasant from the canton of Lucerne in central Switzerland. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757-1810), titled *Costumes de Différent Pays*, published in France in 1797. Each illustration is finely drawn and colored by hand.

The rich variety of Grasset de Saint–Sauveur's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur's pictures.

Part 1

Getting started

Have you watched a blockbuster on Netflix, bought a gadget on Amazon.com, or booked a room on Airbnb today? If so, you have used Amazon Web Services (AWS) in the background. Because Netflix, Amazon.com, and Airbnb all use Amazon Web Services for their business.

Amazon Web Services is the biggest player in the cloud computing markets. According to analysts, AWS maintains a market share of more than 30%.[1] Another impressive number: AWS reported net sales of $4.1 billion USD for the quarter ending in June 2017.[2] AWS data centers are distributed worldwide in North America, South America, Europe, Asia, and Australia. But the cloud does not consist of hardware and computing power alone. Software is part of every cloud platform and makes the difference for you, as a customer who aims to provide a valuable experience to your services's users. The research firm Gartner has yet again classified AWS as a leader in their Magic Quadrant for Cloud Infrastructure as a Service in 2017. Gartner's Magic Quadrant groups vendors into four quadrants: niche players, challengers, visionaries, and leaders, and provides a quick overview of the cloud computing market.[3] Being recognized as a leader attests AWS's high speed and high quality of innovation.

---

[1] Synergy Research Group, "The Leading Cloud Providers Continue to Run Away with the Market," http://mng.bz/qDYo.

[2] Amazon, 10-Q for Quarter Ended June 30 (2017), http://mng.bz/1LAX.

[3] AWS Blog, "AWS Named as a Leader in Gartner's Infrastructure as a Service (IaaS) Magic Quadrant for 7th Consecutive Year," http://mng.bz/0W1W.

The first part of this book will guide you through your initial steps with AWS. You will get an impression of how you can use AWS to improve your IT infrastructure.

Chapter 1 introduces cloud computing and AWS. This will get you familiar with the big-picture basics of how AWS is structured.

Chapter 2 brings Amazon Web Service into action. Here, you will spin up and dive into a complex cloud infrastructure with ease.

1 What is Amazon Web Services?

This chapter covers

- Overview of Amazon Web Services
- The benefits of using Amazon Web Services
- What you can do with Amazon Web Services
- Creating and setting up an AWS account

Amazon Web Services (AWS) is a platform of web services that offers solutions for computing, storing, and networking, at different layers of abstraction. For example, you can use block-level storage (a low level of abstraction) or a highly distributed object storage (a high level of abstraction) to store your data. You can use these services to host websites, run enterprise applications, and mine tremendous amounts of data. *Web services* are accessible via the internet by using typical web protocols (such as HTTP) and used by machines or by humans through a UI. The most prominent services provided by AWS are EC2, which offers virtual machines, and S3, which offers storage capacity. Services on AWS work well together: you can use them to replicate your existing local network setup, or you can design a new setup from scratch. The pricing model for services is pay-per-use.

As an AWS customer, you can choose among different *data centers*. AWS data centers are distributed worldwide. For example, you can start a virtual machine in Japan in exactly the same way as you would start one in Ireland. This enables you to serve customers worldwide with a global infrastructure.

The map in figure 1.1 shows AWS's data centers. Access is limited to some of them: some data centers are accessible for U.S. government organizations only, and special conditions apply for the data centers in China. Additional data centers have been announced for Bahrain, Hong Kong, Sweden, and the U.S..

* Limited access

Figure 1.1 AWS data center locations

In more general terms, AWS is known as a *cloud computing platform*.

1.1 What is cloud computing?

Almost every IT solution is labeled with the term *cloud computing* or just *cloud* nowadays. Buzzwords like this may help sales, but they're hard to work with in a book. So for the sake of clarity, let's define some terms.

Cloud computing, or the cloud, is a metaphor for supply and consumption of IT resources. The IT resources in the cloud aren't directly visible to the user; there are layers of abstraction in between. The level of abstraction offered by the cloud varies, from offering virtual machines (VMs) to providing software as a service (SaaS) based on complex distributed systems. Resources are available on demand in enormous quantities, and you pay for what you use.

The official definition from the National Institute of Standards and Technology:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (networks, virtual machines, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*
>
> —National Institute of Standards and Technology, *The NIST Definition of Cloud Computing*

Clouds are often divided into three types:

- *Public*—A cloud managed by an organization and open to use by the general public.
- *Private*—A cloud that virtualizes and distributes the IT infrastructure for a single organization.
- *Hybrid*—A mixture of a public and a private cloud.

AWS is a public cloud. Cloud computing services also have several classifications:

- *Infrastructure as a service (IaaS)*—Offers fundamental resources like computing, storage, and networking capabilities, using virtual machines such as Amazon EC2, Google Compute Engine, and Microsoft Azure.
- *Platform as a service (PaaS)*—Provides platforms to deploy custom applications to the cloud, such as AWS Elastic Beanstalk, Google App Engine, and Heroku.
- *Software as a service (SaaS)*—Combines infrastructure and software running in the cloud, including office applications like Amazon WorkSpaces, Google Apps for Work, and Microsoft Office 365.

The AWS product portfolio contains IaaS, PaaS, and SaaS. Let's take a more concrete look at what you can do with AWS.

1.2 What can you do with AWS?

You can run all sorts of application on AWS by using one or a combination of services. The examples in this section will give you an idea of what you can do.

1.2.1 Hosting a web shop

John is CIO of a medium-sized e-commerce business. He wants to develop a fast and reliable web shop. He initially decided to host the web shop on-premises, and three years ago he rented machines in a data center. A web server handles requests from customers, and a database stores product information and orders. John is evaluating how his company can take advantage of AWS by running the same setup on AWS, as shown in figure 1.2.

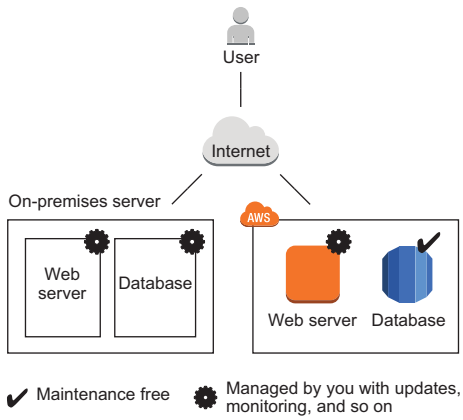


Figure 1.2 Running a web shop on-premises vs. on AWS

John not only wants to lift-and-shift his current on-premises infrastructure to AWS; he wants to get the most out of the advantages the cloud is offering. Additional AWS services allow John to improve his setup.

- The web shop consists of dynamic content (such as products and their prices) and static content (such as the company logo). Splitting these up would reduce the load on the web servers and improve performance by delivering the static content over a content delivery network (CDN).
- Switching to maintenance-free services including a database, an object store, and a DNS system would free John from having to manage these parts of the system, decreasing operational costs and improving quality.
- The application running the web shop can be installed on virtual machines. Using AWS, John can run the same amount of resources he was using on his on-premises machine, but split into multiple smaller virtual machines at no extra cost. If one of these virtual machines fails, the load balancer will send customer requests to the other virtual machines. This setup improves the web shop's reliability.

Figure 1.3 shows how John enhanced the web shop setup with AWS.

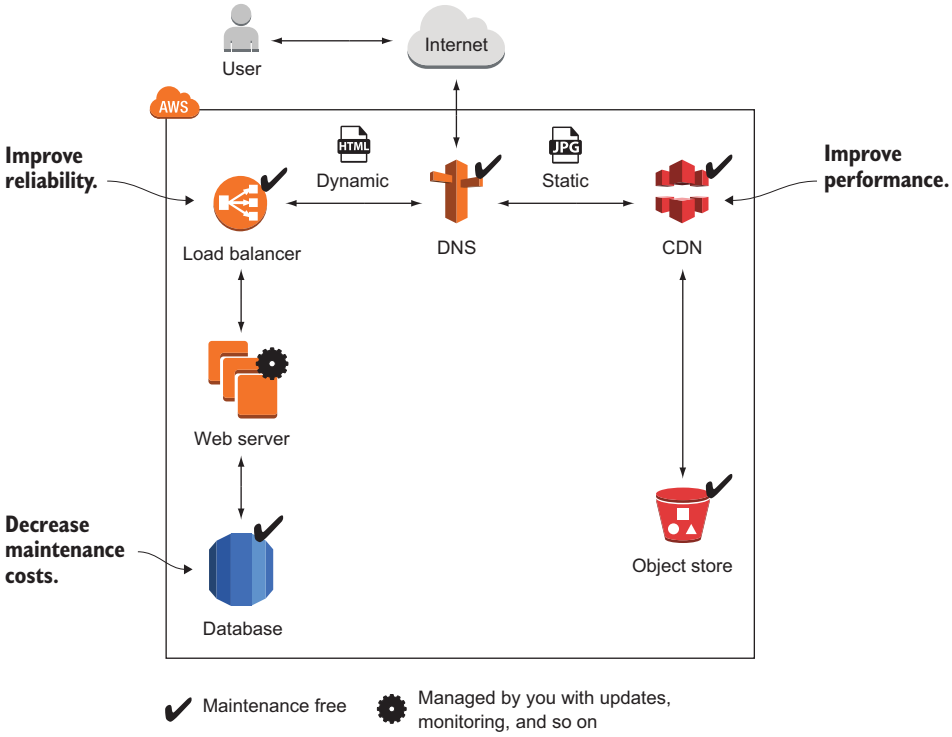


Figure 1.3 Running a web shop on AWS with CDN for better performance, a load balancer for high availability, and a managed database to decrease maintenance costs

John is happy with running his web shop on AWS. By migrating his company's infrastructure to the cloud, he was able to increase the reliability and performance of the web shop.

1.2.2 Running a Java EE application in your private network

Maureen is a senior system architect in a global corporation. She wants to move parts of her company's business applications to AWS when the data-center contract expires in a few months, to reduce costs and gain flexibility. She wants to run enterprise applications (such as Java EE applications) consisting of an application server and an SQL database on AWS. To do so, she defines a virtual network in the cloud and connects it to the corporate network through a Virtual Private Network (VPN) connection. She installs application servers on virtual machines to run the Java EE application. Maureen also wants to store data in an SQL database service (such as Oracle Database Enterprise Edition or Microsoft SQL Server EE).

For security, Maureen uses subnets to separate systems with different security levels from each other. By using access-control lists, she can control ingoing and outgoing traffic for each subnet. For example, the database is only accessible from the JEE server's subnet which helps to protect mission-critical data. Maureen controls traffic to the internet by using Network Address Translation (NAT) and firewall rules as well. Figure 1.4 illustrates Maureen's architecture.

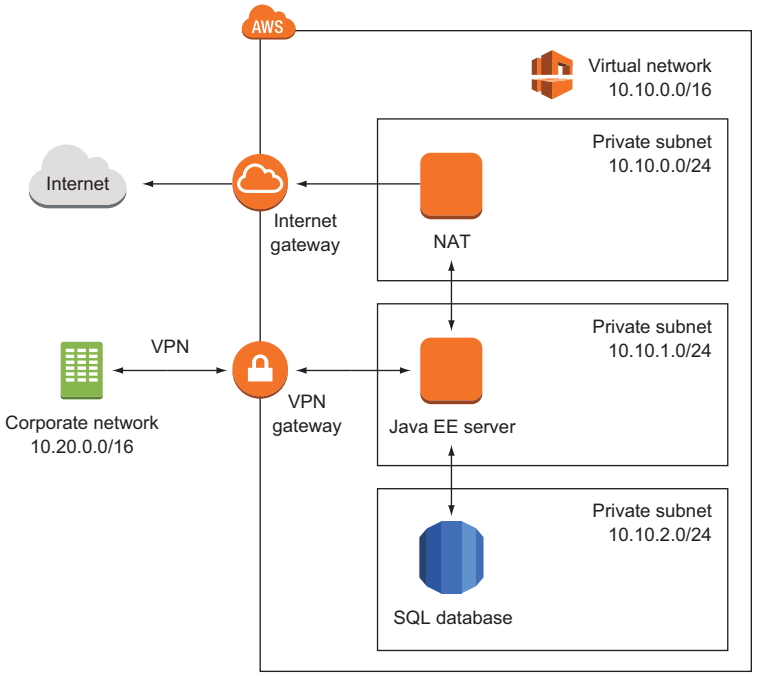


Figure 1.4 Running a Java EE application with enterprise networking on AWS improves flexibility and lowers costs.

Maureen has managed to connect the local data center with a private network running remotely on AWS to enable clients to access the JEE server. To get started, Maureen uses a VPN connection between the local data center and AWS, but she is already thinking about setting up a dedicated network connection to reduce network costs and increase network throughput in the future.

The project was a great success for Maureen. She was able to reduce the time needed to set up an enterprise application from months to hours, as AWS can take care of the virtual machines, databases, and even the networking infrastructure on demand within a few minutes. Maureen's project also benefits from lower infrastructure costs on AWS, compared to using their own infrastructure on-premises.

1.2.3 Implementing a highly available system

Alexa is a software engineer working for a fast-growing startup. She knows that Murphy's Law applies to IT infrastructure: anything that can go wrong *will* go wrong. Alexa is working hard to build a highly available system to prevent outages from ruining the business. All services on AWS are either highly available or can be used in a highly available way. So, Alexa builds a system like the one shown in figure 1.5 with a high availability architecture. The database service is offered with replication and fail-over handling. In case the master database instance fails, the standby database is promoted as the new master database automatically. Alexa uses virtual machines acting as web servers. These virtual machines aren't highly available by default, but Alexa launches multiple virtual machines in different data centers to achieve high availability. A load balancer checks the health of the web servers and forwards requests to healthy machines.

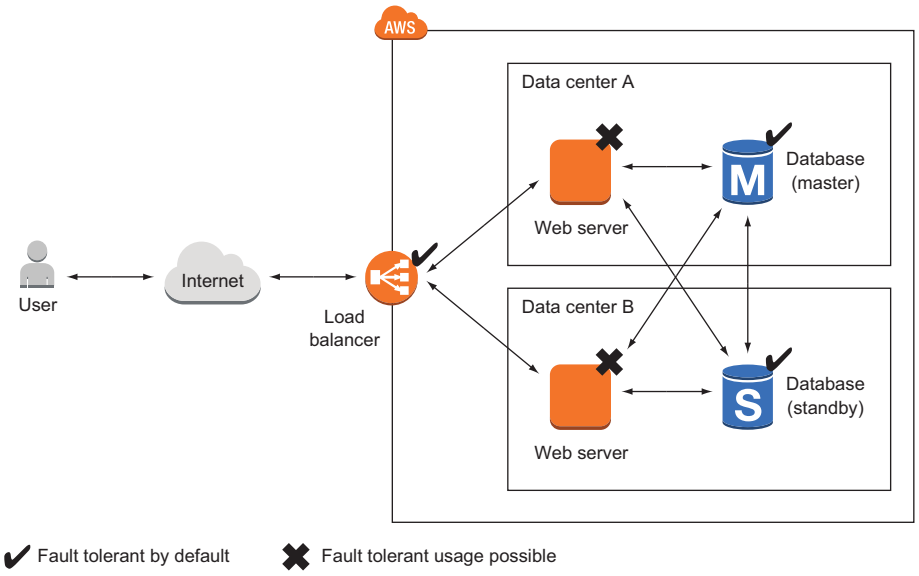


Figure 1.5 Building a highly available system on AWS by using a load balancer, multiple virtual machines, and a database with master-standby replication

So far, Alexa has protected the startup from major outages. Nevertheless, she and her team are always planning for failure and are constantly improving the resilience of their systems.

1.2.4 Profiting from low costs for batch processing infrastructure

Nick is a data scientist who needs to process massive amounts of measurement data collected from gas turbines. He needs to generate a report containing the maintenance condition of hundreds of turbines daily. Therefore, his team needs a computing infrastructure to analyze the newly arrived data once a day. Batch jobs are run on a schedule and store aggregated results in a database. A business intelligence (BI) tool is used to generate reports based on the data stored in the database.

As the budget for computing infrastructure is very small, Nick and his team have been looking for a cost effective solution to analyze their data. He finds a way to make clever use of AWS's price model:

- *AWS bills virtual machines per minute.* So Nick launches a virtual machine when starting a batch job, and terminates it immediately after the job finished. Doing so allows him to pay for computing infrastructure only when actually using it. This is a big game changer compared to the traditional data center where Nick had to pay a monthly fee for each machine, no matter how much it was used.
- *AWS offers spare capacity in their data centers at substantial discount.* It is not important for Nick to run a batch job at a specific time. He can wait to execute a batch job until there is enough spare capacity available, so AWS offers him a virtual machine with a discount of 50%.

Figure 1.6 illustrates how Nick benefits from the pay-per-use price model for virtual machines.

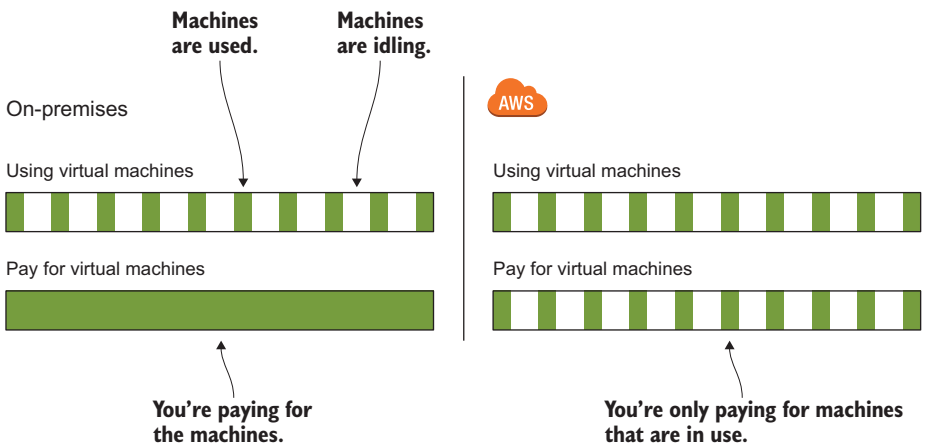


Figure 1.6 Making use of the pay-per-use price model of virtual machines

Nick is happy to have access to a computing infrastructure that allows his team to analyze data at low costs. You now have a broad idea of what you can do with AWS. Generally speaking, you can host any application on AWS. The next section explains the nine most important benefits AWS has to offer.

1.3 How you can benefit from using AWS

What's the most important advantage of using AWS? Cost savings, you might say. But saving money isn't the only advantage. Let's look at how else you can benefit from using AWS.

1.3.1 Innovative and fast-growing platform

AWS is announcing new services, features, and improvements constantly. Go to https://aws.amazon.com/about-aws/whats-new/ to get an impression of the speed of innovation. We have counted 719 announcements from Jan. 1 to Oct. 21 in 2017, and 641 announcements in 2016. Making use of the innovative technologies provided by AWS helps you to generate valuable solutions for your customers and thus achieve a competitive advantage.

AWS reported net sales of $4.1 billion USD for the quarter ending in June 2017. That's a year-over-year growth rate of 42% (Q3 2016 versus Q3 2017). We expect AWS to expand the size and extend of its platform in the upcoming years, for example, by adding additional services and data centers.[4]

1.3.2 Services solve common problems

As you've learned, AWS is a platform of services. Common problems such as load balancing, queuing, sending email, and storing files are solved for you by services. You don't need to reinvent the wheel. It's your job to pick the right services to build complex systems. So let AWS manage those services while you focus on your customers.

1.3.3 Enabling automation

Because AWS has an API, you can automate everything: you can write code to create networks, start virtual machine clusters, or deploy a relational database. Automation increases reliability and improves efficiency.

The more dependencies your system has, the more complex it gets. A human can quickly lose perspective, whereas a computer can cope with graphs of any size. You should concentrate on tasks humans are good at—such as describing a system—while the computer figures out how to resolve all those dependencies to create the system. Setting up an environment in the cloud based on your blueprints can be automated with the help of infrastructure as code, covered in chapter 4.

---

[4] Amazon, 10-Q for Quarter Ended June 30 (2017), http://mng.bz/1LAX.

1.3.4 Flexible capacity (scalability)

Flexible capacity frees you from planning. You can scale from one virtual machine to thousands of virtual machines. Your storage can grow from gigabytes to petabytes. You no longer need to predict your future capacity needs for the coming months and years.

If you run a web shop, you have seasonal traffic patterns, as shown in figure 1.7. Think about day versus night, and weekday versus weekend or holiday. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? That's exactly what flexible capacity is about. You can start new virtual machines within minutes and throw them away a few hours after that.

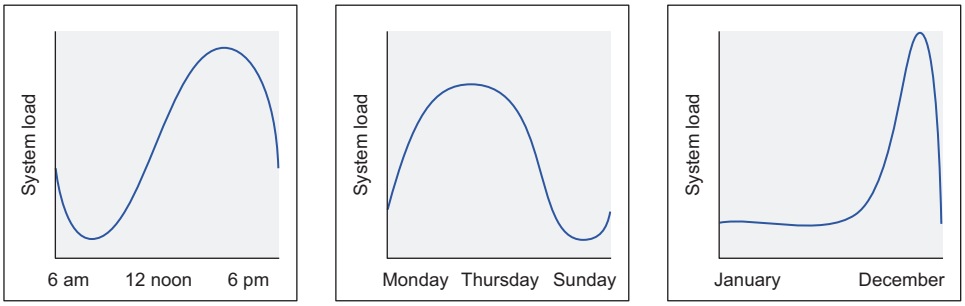


Figure 1.7 Seasonal traffic patterns for a web shop

The cloud has almost no capacity constraints. You no longer need to think about rack space, switches, and power supplies—you can add as many virtual machines as you like. If your data volume grows, you can always add new storage capacity.

Flexible capacity also means you can shut down unused systems. In one of our last projects, the test environment only ran from 7 a.m. to 8 p.m. on weekdays, allowing us to save 60%.

1.3.5 Built for failure (reliability)

Most AWS services are highly available or fault tolerant by default. If you use those services, you get reliability for free. AWS supports you as you build systems in a reliable way. It provides everything you need to create your own highly available or fault-tolerant systems.

1.3.6 Reducing time to market

In AWS, you request a new virtual machine, and a few minutes later that virtual machine is booted and ready to use. The same is true with any other AWS service available. You can use them all on demand.

Your development process will be faster because of the shorter feedback loops. You can eliminate constraints such as the number of test environments available; if you need another test environment, you can create it for a few hours.

1.3.7 Benefiting from economies of scale

AWS is increasing its global infrastructure constantly. Thus AWS benefits from an economy of scale. As a customer, you will benefit partially from these effects.

AWS reduces prices for their cloud services every now and then. A few examples:

- In November 2016, charges for storing data on the object storage S3 were reduced by 16% to 28%.
- In May 2017, prices were reduced by 10% to 17% for virtual machines with a one- or three-year commitment (reserved instances).
- In July 2017, AWS reduced prices for virtual machines running a Microsoft SQL Server (Standard Edition) by up to 52%.

1.3.8 Global infrastructure

Are you serving customers worldwide? Making use of AWS's global infrastructure has the following advantages: low network latencies between your customers and your infrastructure, being able to comply with regional data protection requirements, and benefiting from different infrastructure prices in different regions. AWS offers data centers in North America, South America, Europe, Asia, and Australia, so you can deploy your applications worldwide with little extra effort.

1.3.9 Professional partner

When you use AWS services, you can be sure that their quality and security follow the latest standards and certifications. For example:

- *ISO 27001*—A worldwide information security standard certified by an independent and accredited certification body.
- *ISO 9001*—A standardized quality management approach used worldwide and certified by an independent and accredited certification body.
- *PCI DSS Level 1*—A data security standard (DSS) for the payment card industry (PCI) to protect cardholders data.

Go to https://aws.amazon.com/compliance/ if you want to dive into the details. If you're still not convinced that AWS is a professional partner, you should know that Expedia, Vodafone, FDA, FINRA, Airbnb, Slack, and many more are running serious workloads on AWS.[5]

We have discussed a lot of reasons to run your workloads on AWS. But what does AWS cost? You will learn more about the pricing models in the next section.

1.4 How much does it cost?

A bill from AWS is similar to an electric bill. Services are billed based on use. You pay for the time a virtual machine was running, the used storage from the object store, or the number of running load balancers. Services are invoiced on a monthly basis. The

[5] AWS Customer Success, https://aws.amazon.com/solutions/case-studies/.

pricing for each service is publicly available; if you want to calculate the monthly cost of a planned setup, you can use the AWS Simple Monthly Calculator (http://aws.amazon.com/calculator).

1.4.1 Free Tier

You can use some AWS services for free within the first 12 months of your signing up. The idea behind the Free Tier is to enable you to experiment with AWS and get some experience using its services. Here is a taste of what's included in the Free Tier:

- 750 hours (roughly a month) of a small virtual machine running Linux or Windows. This means you can run one virtual machine for a whole month or you can run 750 virtual machines for one hour.
- 750 hours (or roughly a month) of a classic or application load balancer.
- Object store with 5 GB of storage.
- Small database with 20 GB of storage, including backup.

If you exceed the limits of the Free Tier, you start paying for the resources you consume without further notice. You'll receive a bill at the end of the month. We'll show you how to monitor your costs before you begin using AWS.

After your one-year trial period ends, you pay for all resources you use. But some resources are free forever. For example, the first 25 GB of the NoSQL database are free forever.

You get additional benefits, as detailed at http://aws.amazon.com/free. This book will use the Free Tier as much as possible and will clearly state when additional resources are required that aren't covered by the Free Tier.

1.4.2 Billing example

As mentioned earlier, you can be billed in several ways:

- *Based on minutes or hours of usage*—A virtual machine is billed per minute. A load balancer is billed per hour.
- *Based on traffic*—Traffic is measured in gigabytes or in number of requests, for example.
- *Based on storage usage*—Usage can be measured by capacity (for example, 50 GB volume no matter how much you use) or real usage (such as 2.3 GB used).

Remember the web shop example from section 1.2? Figure 1.8 shows the web shop and adds information about how each part is billed.

Let's assume your web shop started successfully in January, and you ran a marketing campaign to increase sales for the next month. Lucky you: you were able to increase the number of visitors to your web shop fivefold in February. As you already know, you have to pay for AWS based on usage. Table 1.1 shows your bill for February. The number of visitors increased from 100,000 to 500,000, and your monthly bill increased from $127 USD to $495 USD, which is a 3.9-fold increase. Because your web shop had to handle more traffic, you had to pay more for services, such as the CDN,

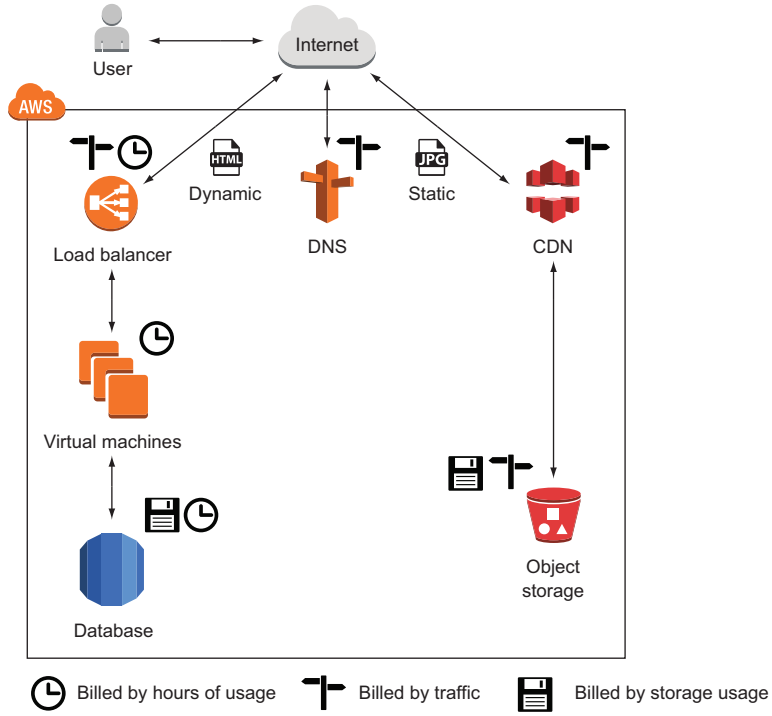


Figure 1.8 AWS bills services on minutes or hours of usage, by traffic, or by used storage.

the web servers, and the database. Other services, like the amount of storage needed for static files, didn't change, so the price stayed the same.

Table 1.1 How an AWS bill changes if the number of web shop visitors increases

Service	January usage	February usage	February charge	Increase
Visits to website	100,000	500,000		
CDN	25 M requests + 25 GB traffic	125 M requests + 125 GB traffic	$135.63 USD	$107.50 USD
Static files	50 GB used storage	50 GB used storage	$1.15 USD	$0.00 USD
Load balancer	748 hours + 50 GB traffic	748 hours + 250 GB traffic	$20.70 USD	$1.60 USD
Web servers	1 virtual machine = 748 hours	4 virtual machines = 2,992 hours	$200.46 USD	$150.35 USD
Database (748 hours)	Small virtual machine + 20 GB storage	Large virtual machine + 20 GB storage	$133.20 USD	$105.47 USD
DNS	2 M requests	10 M requests	$4.00 USD	$3.20 USD
Total cost			***$495.14 USD***	***$368.12 USD***

With AWS, you can achieve a linear relationship between traffic and costs. And other opportunities await you with this pricing model.

1.4.3 Pay-per-use opportunities

The AWS pay-per-use pricing model creates new opportunities. For example, the barrier for starting a new project is lowered, as you no longer need to invest in infrastructure up front. You can start virtual machines on demand and only pay per second of usage, and you can stop using those virtual machines whenever you like and no longer have to pay for them. You don't need to make an upfront commitment regarding how much storage you'll use.

Another example: a big server costs exactly as much as two smaller ones with the same capacity. Thus you can divide your systems into smaller parts, because the cost is the same. This makes fault tolerance affordable not only for big companies but also for smaller budgets.

1.5 Comparing alternatives

AWS isn't the only cloud computing provider. Microsoft Azure and Google Cloud Platform (GCP) are major players as well.

The three major cloud providers share a lot in common. They all have:

- A worldwide infrastructure that provides computing, networking, and storage capabilities.
- An IaaS offering that provides virtual machines on-demand: Amazon EC2, Azure Virtual Machines, Google Compute Engine.
- Highly distributed storage systems able to scale storage and I/O capacity without limits: Amazon S3, Azure Blob storage, Google Cloud Storage.
- A pay-as-you-go pricing model.

But what are the differences between the cloud providers?

AWS is the market leader in cloud computing, offering an extensive product portfolio. Even if AWS has expanded into the enterprise sector during recent years, it is still obvious that AWS started with services to solve internet-scale problems. Overall, AWS is building great services based on innovative, mostly open source, technologies. AWS offers complicated but rock-solid ways to restrict access to your cloud infrastructure.

Microsoft Azure provides Microsoft's technology stack in the cloud, recently expanding into web-centric and open source technologies as well. It seems like Microsoft is putting a lot of effort into catching up with Amazon's market share in cloud computing.

GCP is focused on developers looking to build sophisticated distributed systems. Google combines their worldwide infrastructure to offer scalable and fault-tolerant services (such as Google Cloud Load Balancing). The GCP seems more focused on cloud-native applications than on migrating your locally hosted applications to the cloud, in our opinion.

There are no shortcuts to making an informed decision about which cloud provider to choose. Each use case and project is different. The devil is in the details. Also don't forget where you are coming from. (Are you using Microsoft technology heavily? Do you have a big team consisting of system administrators or are you a developer-centric company?) Overall, in our opinion, AWS is the most mature and powerful cloud platform available at the moment.

1.6 Exploring AWS services

Hardware for computing, storing, and networking is the foundation of the AWS cloud. AWS runs services on this hardware, as shown in figure 1.9. The API acts as an interface between AWS services and your applications.

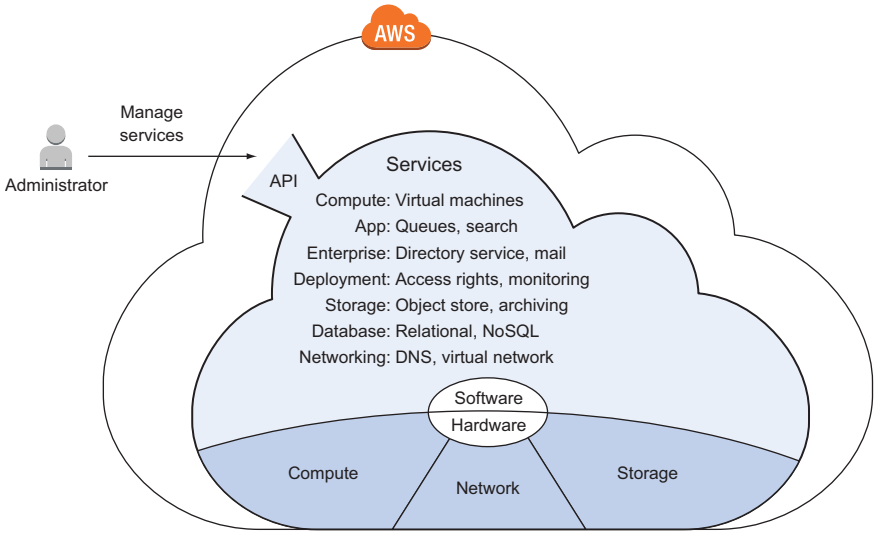


Figure 1.9 The AWS cloud is composed of hardware and software services accessible via an API.

You can manage services by sending requests to the API manually via a web-based UI like the Management Console, a command-line interface (CLI), or programmatically via an SDK. Virtual machines have a special feature: you can connect to virtual machines through SSH, for example, and gain administrator access. This means you can install any software you like on a virtual machine. Other services, like the NoSQL database service, offer their features through an API and hide everything that's going on behind the scenes. Figure 1.10 shows an administrator installing a custom PHP web application on a virtual machine and managing dependent services such as a NoSQL database used by the application.

Users send HTTP requests to a virtual machine. This virtual machine is running a web server along with a custom PHP web application. The web application needs to talk to AWS services in order to answer HTTP requests from users. For example, the

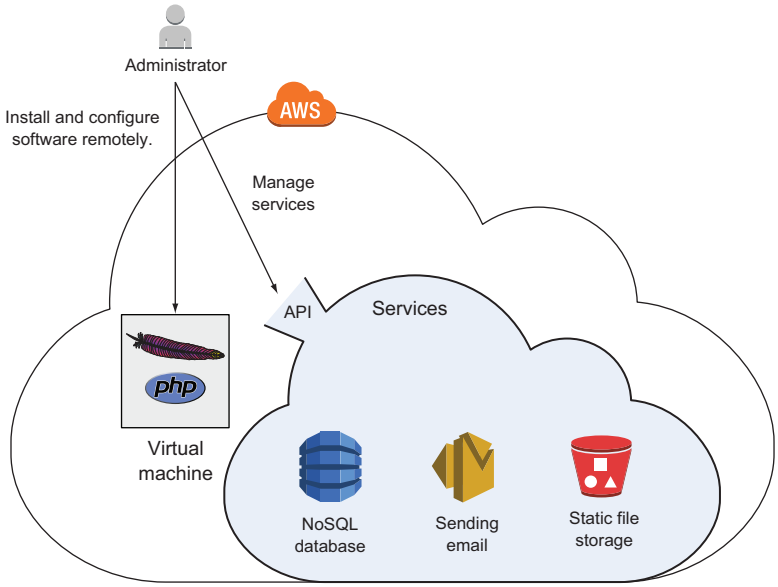


Figure 1.10 Managing a custom application running on a virtual machine and dependent services

application might need to query data from a NoSQL database, store static files, and send email. Communication between the web application and AWS services is handled by the API, as figure 1.11 shows.

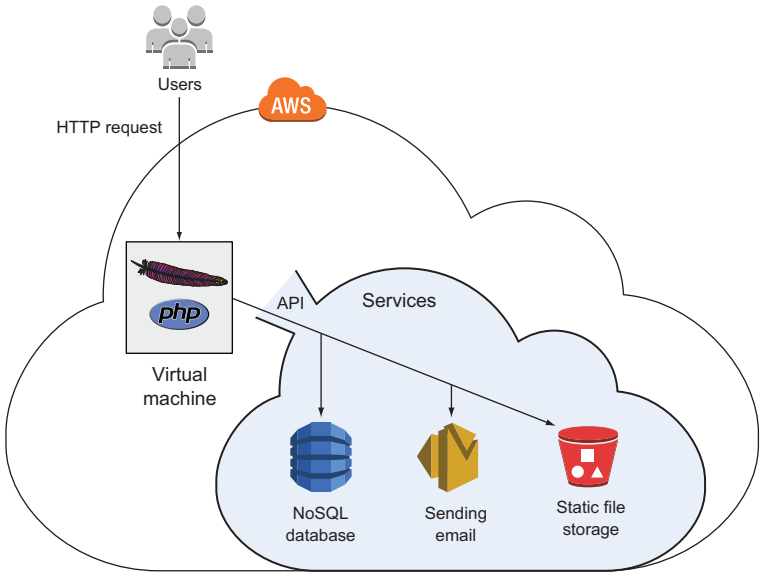


Figure 1.11 Handling an HTTP request with a custom web application using additional AWS services

The number of services available can be scary at the outset. When logging into AWS's web interface you are presented with an overview listing 98 services. On top of that, new services are announced constantly during the year and at the big conference in Las Vegas, AWS re:Invent.

AWS offers services in the following categories:

- Analytics
- Application Integration
- AR and VR
- Business Productivity
- Compute
- Customer Engagement
- Database
- Desktop and App Streaming
- Developer Tools
- Game Development
- Internet Of Things
- Machine Learning
- Management Tools
- Media Services
- Migration
- Mobile Services
- Networking and Content Delivery
- Security, Identity, and Compliance
- Storage

Unfortunately, it is not possible to cover all services offered by AWS in our book. Therefore, we are focusing on the services that will best help you get started quickly, as well as the most widely used services. The following services are covered in detail in our book:

- *EC2*—Virtual machines
- *ELB*—Load balancers
- *Lambda*—Executing functions
- *Elastic Beanstalk*—Deploying web applications
- *S3*—Object store
- *EFS*—Network filesystem
- *Glacier*—Archiving data
- *RDS*—SQL databases
- *DynamoDB*—NoSQL database
- *ElastiCache*—In-memory key-value store
- *VPC*—Private network
- *CloudWatch*—Monitoring and logging
- *CloudFormation*—Automating your infrastructure
- *OpsWorks*—Deploying web applications
- *IAM*—Restricting access to your cloud resources
- *Simple Queue Service*—Distributed queues

We are missing at least three important topics that would fill their own books: continuous delivery, Docker/containers, and Big Data. Let us know when you are interested in reading one of these unwritten books.

But how do you interact with an AWS service? The next section explains how to use the web interface, the CLI, and SDKs to manage and access AWS resources.

1.7 Interacting with AWS

When you interact with AWS to configure or use services, you make calls to the API. The API is the entry point to AWS, as figure 1.12 demonstrates.

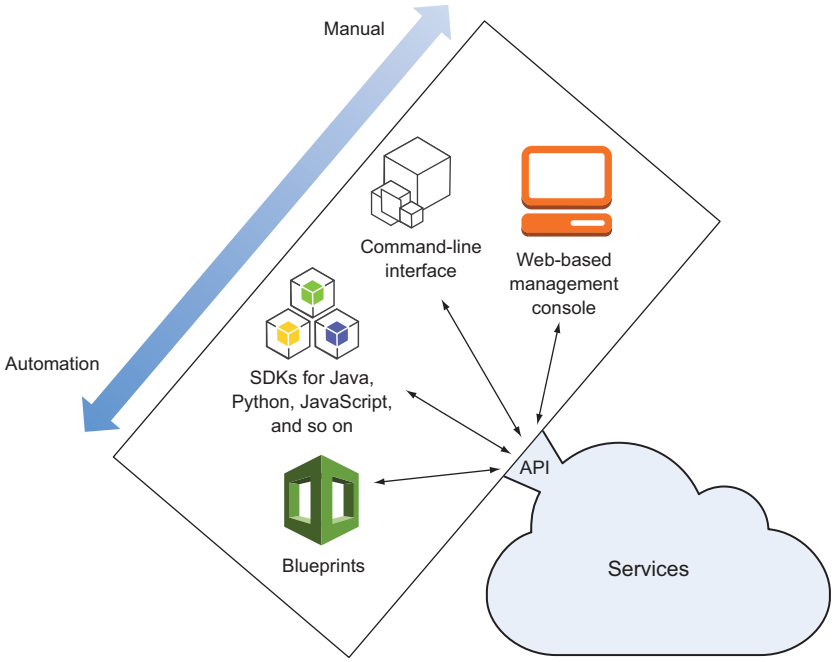


Figure 1.12 Different ways to access the AWS API, allowing you to manage and access AWS services

Next, we'll give you an overview of the tools available for communicating with API: the Management Console, the command-line interface, the SDKs, and infrastructure blueprints. We will compare the different tools, and you will learn how to use all of them while working your way through the book.

1.7.1 Management Console

The AWS Management Console allows you to manage and access AWS services through a graphical user interface (GUI), which runs in every modern web browser (the latest three versions of Google Chrome and Mozilla Firefox; Apple Safari: 9, 8, and 7; Microsoft Internet Explorer: 11; Microsoft Edge: 12). See figure 1.13.

When getting started or experimenting with AWS, the Management Console is the best place to start. It helps you to gain an overview of the different services quickly. The Management Console is also a good way to set up a cloud infrastructure for development and testing.

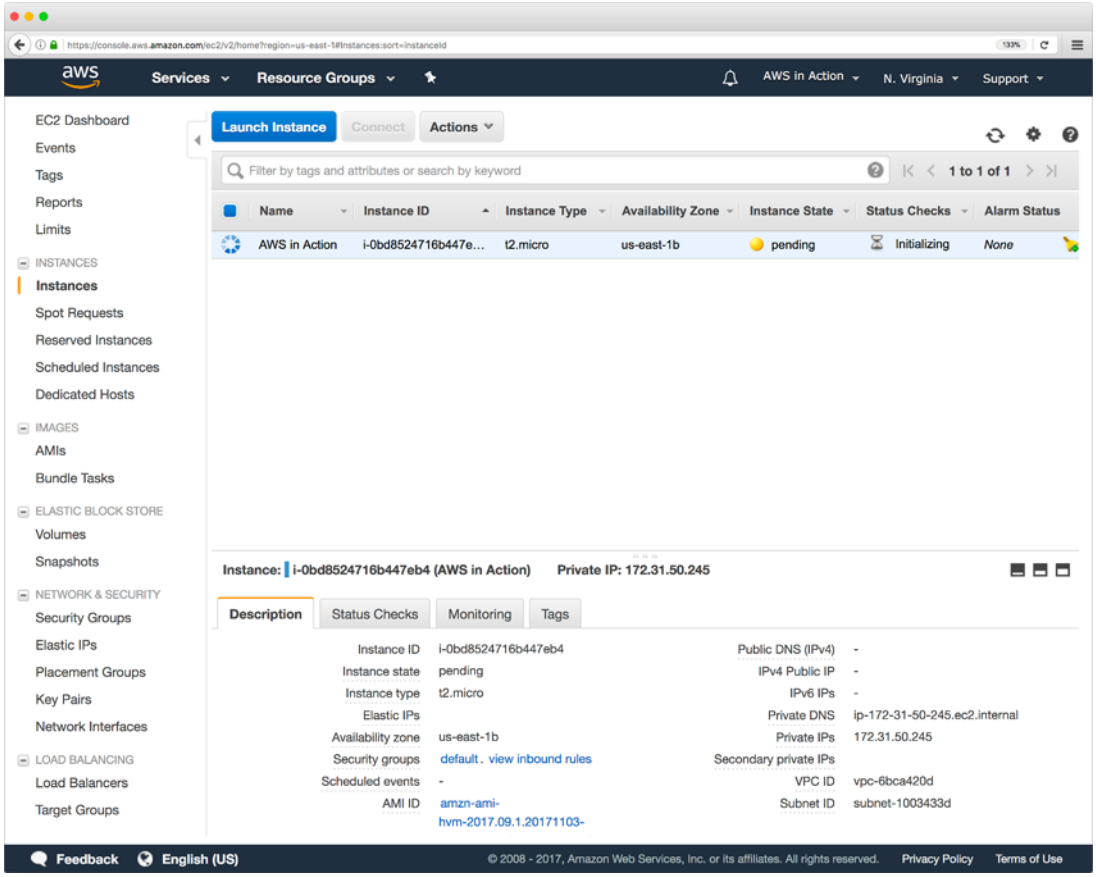


Figure 1.13 The AWS Management Console offers a GUI to manage and access AWS services.

1.7.2 Command-line interface

The command-line interface (CLI) allows you to manage and access AWS services within your terminal. Because you can use your terminal to automate or semi-automate recurring tasks, CLI is a valuable tool. You can use the terminal to create new cloud infrastructures based on blueprints, upload files to the object store, or get the details of your infrastructure's networking configuration regularly. Figure 1.14 shows the CLI in action.

If you want to automate parts of your infrastructure with the help of a continuous integration server, like Jenkins, the CLI is the right tool for the job. The CLI offers a convenient way to access the API and combine multiple calls into a script.

You can even begin to automate your infrastructure with scripts by chaining multiple CLI calls together. The CLI is available for Windows, Mac, and Linux, and there is also a PowerShell version available.

Figure 1.14 The CLI allows you to manage and access AWS services from your terminal.

1.7.3 SDKs

Use your favorite programming language to interact with the AWS API. AWS offers SDKs for the following platforms and languages:

- Android
- Browsers (JavaScript)
- iOS
- Java
- .NET
- Node.js (JavaScript)
- PHP
- Python
- Ruby
- Go
- C++

SDKs are typically used to integrate AWS services into applications. If you're doing software development and want to integrate an AWS service like a NoSQL database or a push-notification service, an SDK is the right choice for the job. Some services, such as queues and topics, must be used with an SDK.

1.7.4 Blueprints

A *blueprint* is a description of your system containing all resources and their dependencies. An Infrastructure as Code tool compares your blueprint with the current system, and calculates the steps to create, update, or delete your cloud infrastructure. Figure 1.15 shows how a blueprint is transferred into a running system.

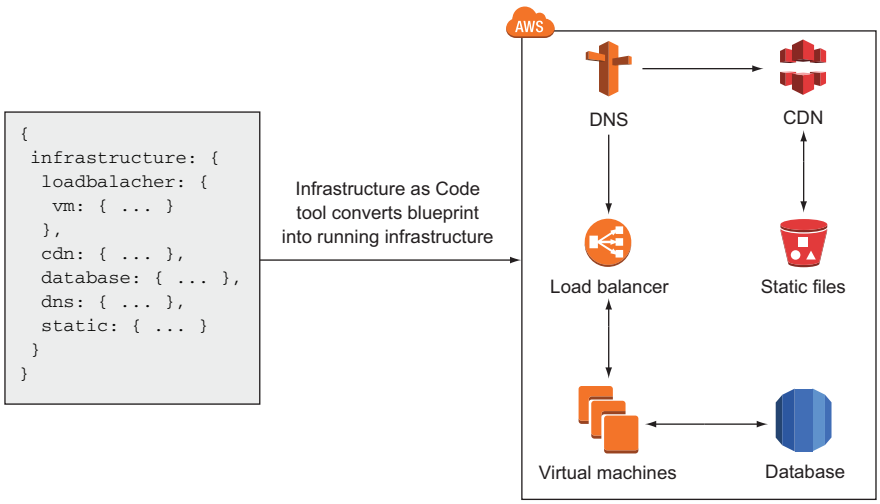


Figure 1.15 Infrastructure automation with blueprints

Consider using blueprints if you have to control many or complex environments. Blueprints will help you to automate the configuration of your infrastructure in the cloud. You can use them to set up a network and launch virtual machines, for example.

Automating your infrastructure is also possible by writing your own source code with the help of the CLI or the SDKs. But doing so requires you to resolve dependencies, make sure you are able to update different versions of your infrastructure, and handle errors yourself. As you will see in chapter 4, using a blueprint and an Infrastructure-as-Code tool solves these challenges for you. It's time to get started creating your AWS account and exploring AWS practice after all that theory.

1.8 Creating an AWS account

Before you can start using AWS, you need to create an account. Your account is a basket for all your cloud resources. You can attach multiple users to an account if multiple humans need access to it; by default, your account will have one root user. To create an account, you need the following:

- A telephone number to validate your identity
- A credit card to pay your bills

USING AN OLD ACCOUNT? It is possible to use your existing AWS account while working through this book. In this case, your usage might not be covered by the Free Tier. So you might have to pay for the use.

If you created your existing AWS account before Dec. 4, 2013, please create a new one, as there are some legacy issues that might cause trouble during our examples.

1.8.1 Signing up

The sign-up process consists of five steps:

1. Provide your login credentials.
2. Provide your contact information.
3. Provide your payment details.
4. Verify your identity.
5. Choose your support plan.

Point your favorite web browser to https://aws.amazon.com, and click the *Create a Free Account* button.

1. Providing your login credentials

Creating an AWS account starts with defining a unique AWS account name, as shown in figure 1.16. The AWS account name has to be globally unique among all AWS customers. Try `aws-in-action-$yourname` and replace `$yourname` with your name. Beside the account name, you have to specify an email address and a password used to authenticate the root user of your AWS account.

We advise you to choose a strong password to prevent misuse of your account. *Use a password consisting of at least 20 characters.* Protecting your AWS account from unwanted access is crucial to avoid data breaches, data loss, or unwanted resource usage on your behalf.

Figure 1.16 Creating an AWS account: sign-up page

2. PROVIDING YOUR CONTACT INFORMATION

The next step, as shown in figure 1.17, is adding your contact information. Fill in all the required fields, and continue.

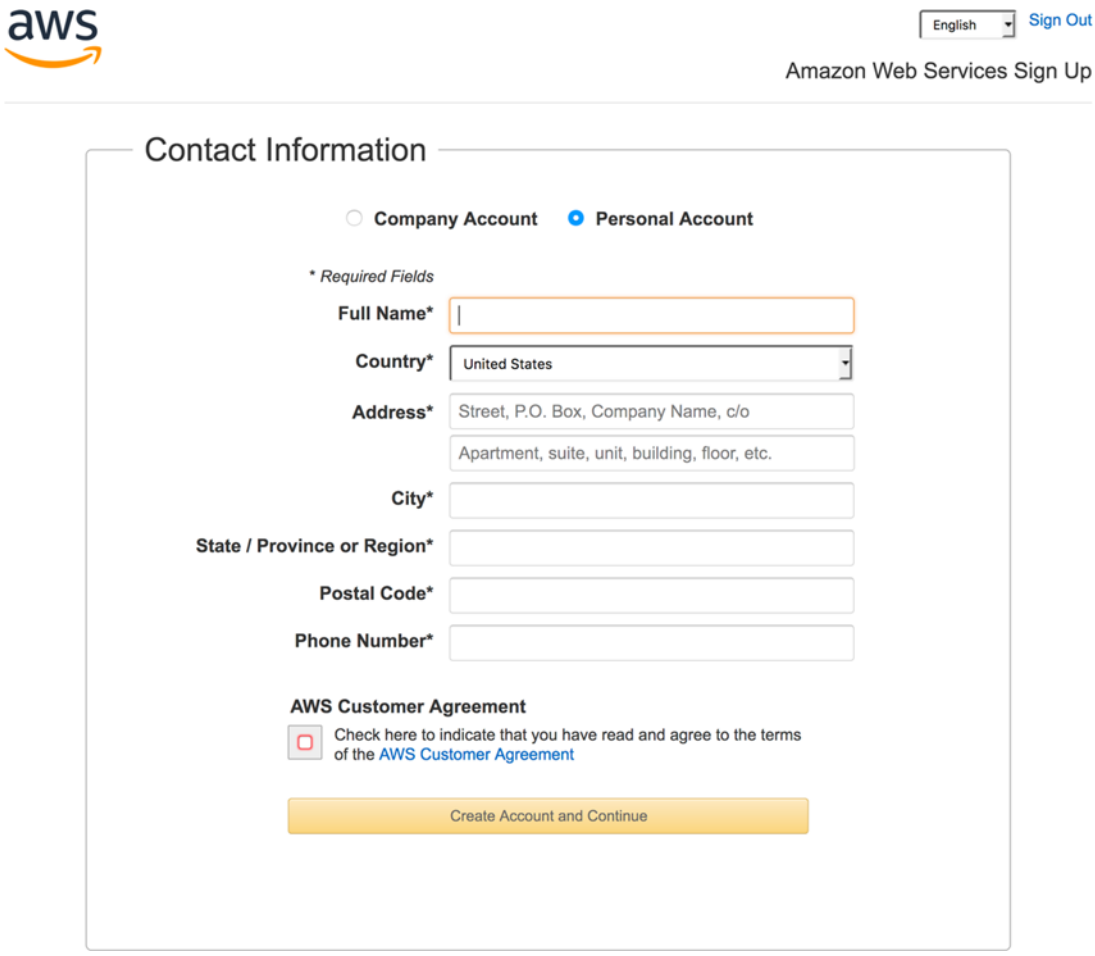


Figure 1.17 Creating an AWS account: providing your contact information

3. PROVIDING YOUR PAYMENT DETAILS

Next the screen shown in figure 1.18 asks for your payment information. Provide your credit card information. There's an option to change the currency setting from USD to AUD, CAD, CHF, DKK, EUR, GBP, HKD, JPY, NOK, NZD, SEK, or ZAR later on if that's more convenient for you. If you choose this option, the amount in USD is converted into your local currency at the end of the month.

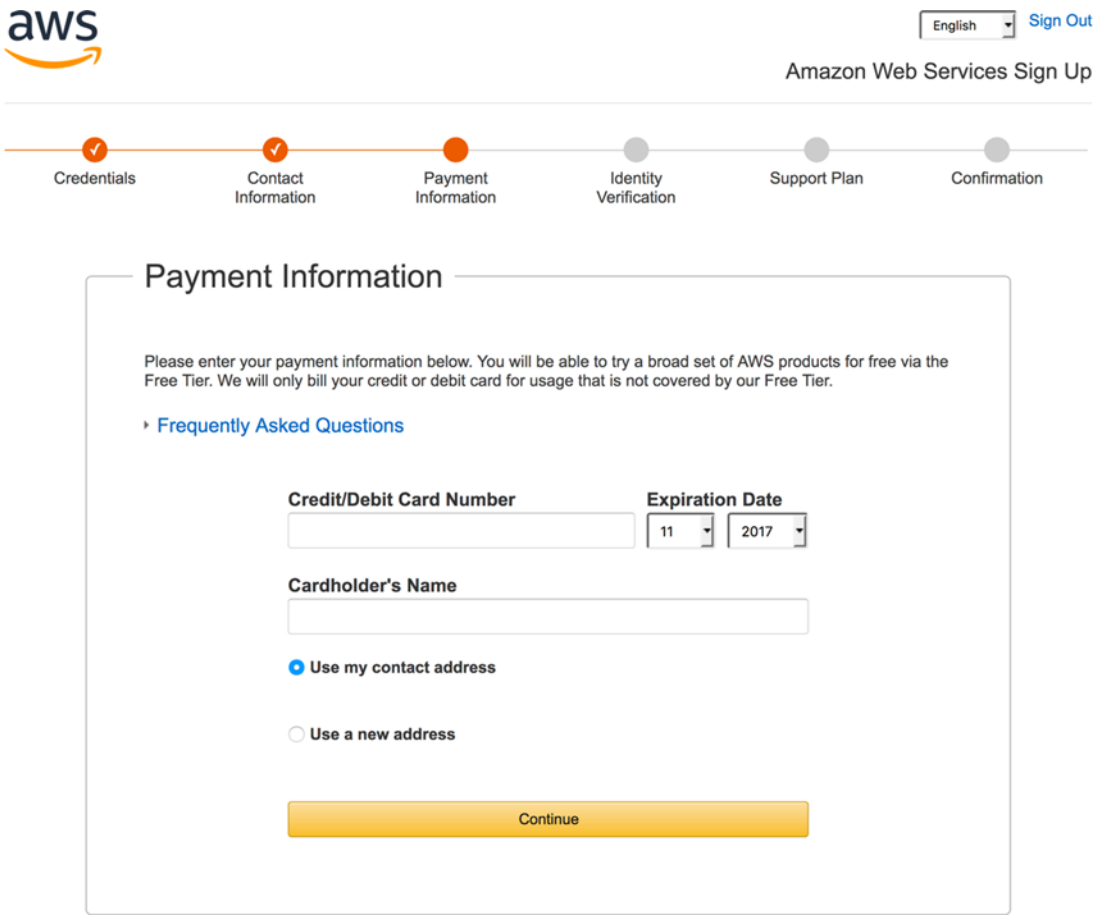


Figure 1.18 Creating an AWS account: providing your payment details

4. Verifying your identity

The next step is to verify your identity. Figure 1.19 shows the first step of the process. After you complete the first part of the form, you'll receive a call from AWS. A robot voice will ask for your PIN. The four-digit PIN is displayed on the website and you have to enter it using your telephone. After your identity has been verified, you are ready to continue with the last step.

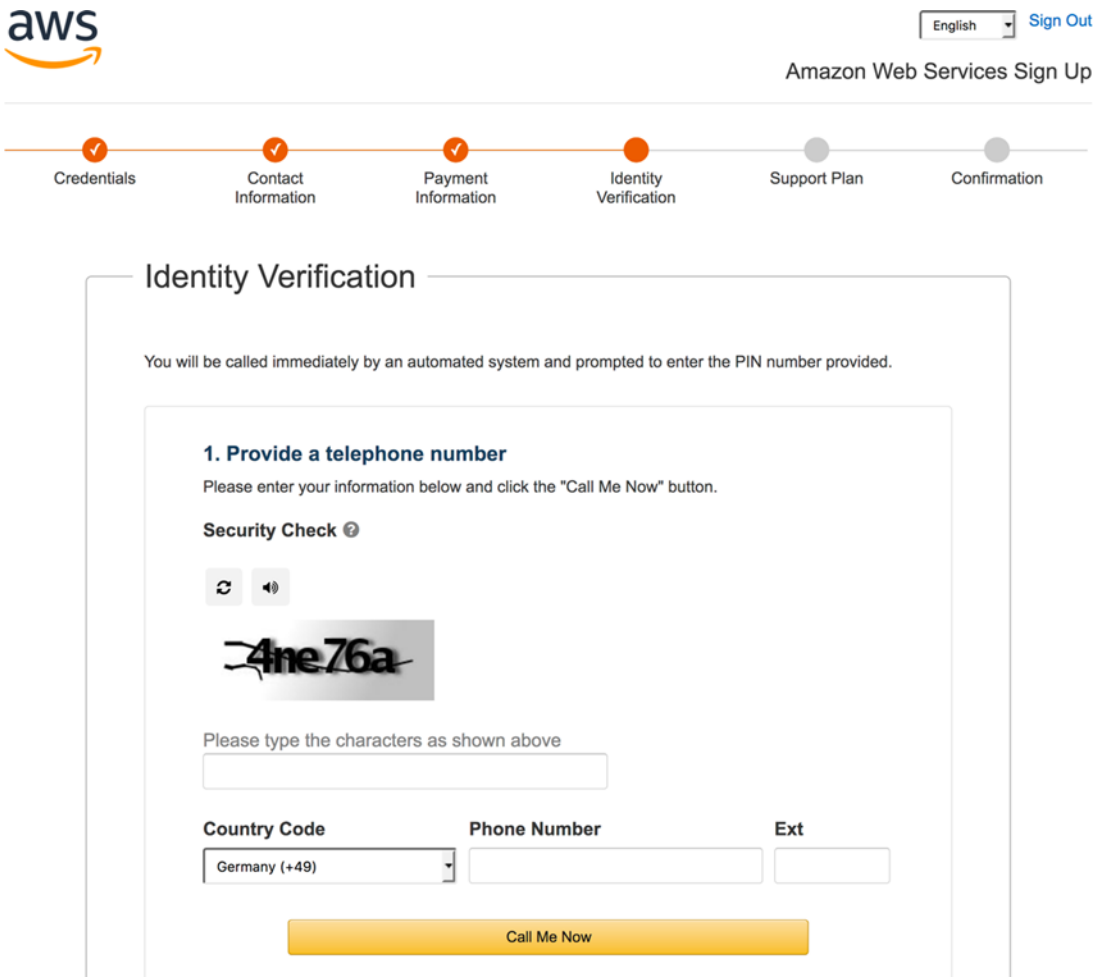


Figure 1.19 Creating an AWS account: verifying your identity

5. CHOOSING YOUR SUPPORT PLAN

The last step is to choose a support plan; see figure 1.20. In this case, select the Basic plan, which is free. If you later create an AWS account for your business, we recommend the Business support plan. You can even switch support plans later.

High five! You're done. Click Launch Management Console as shown in figure 1.21 to sign into your AWS account for the first time.

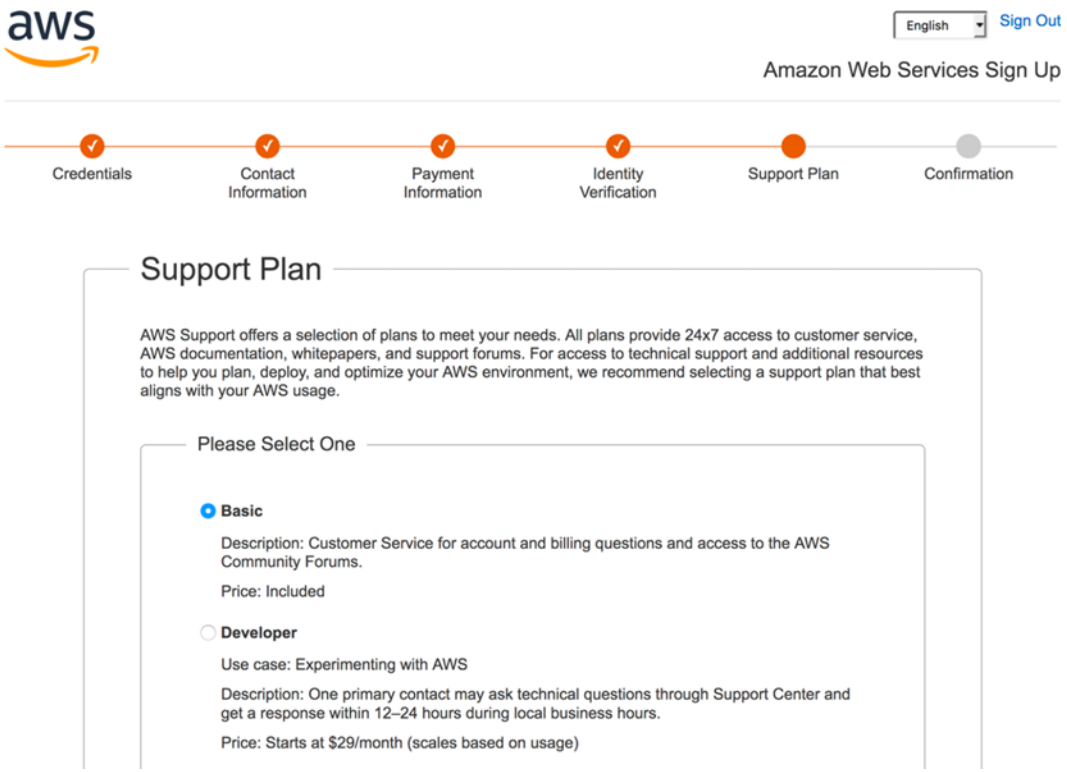


Figure 1.20 Creating an AWS account: choosing your support plan

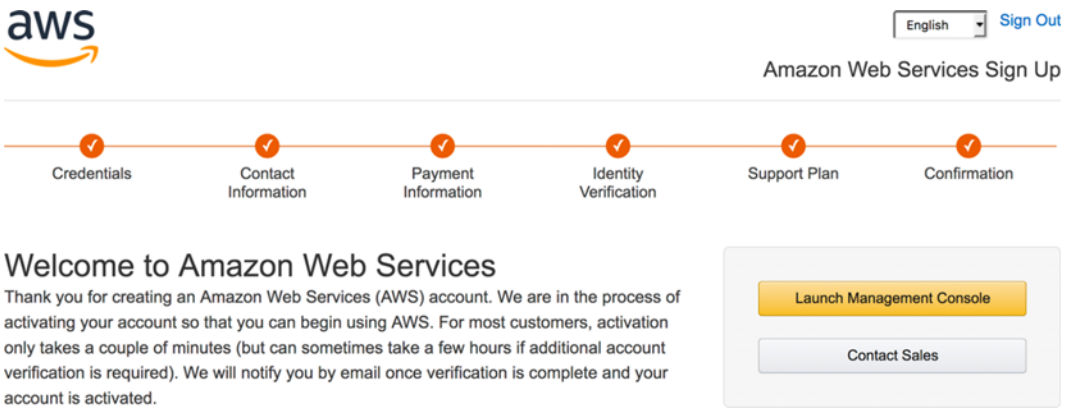


Figure 1.21 You have successfully created an AWS account.

1.8.2 Signing In

You now have an AWS account and are ready to sign in to the AWS Management Console. As mentioned earlier, the Management Console is a web-based tool you can use to control AWS resources; it makes most of the functionality of the AWS API available to you. Figure 1.22 shows the sign-in form at https://console.aws.amazon.com. Enter your email address, click Next, and then enter your password to sign in.

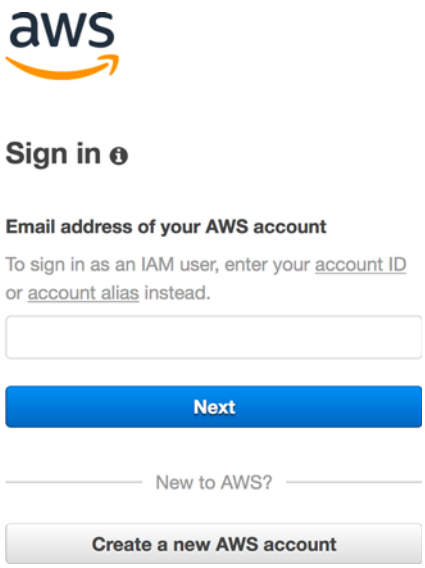


Figure 1.22 Sign in to the Management Console.

After you have signed in successfully, you are forwarded to the start page of the Management Console, as shown in figure 1.23.

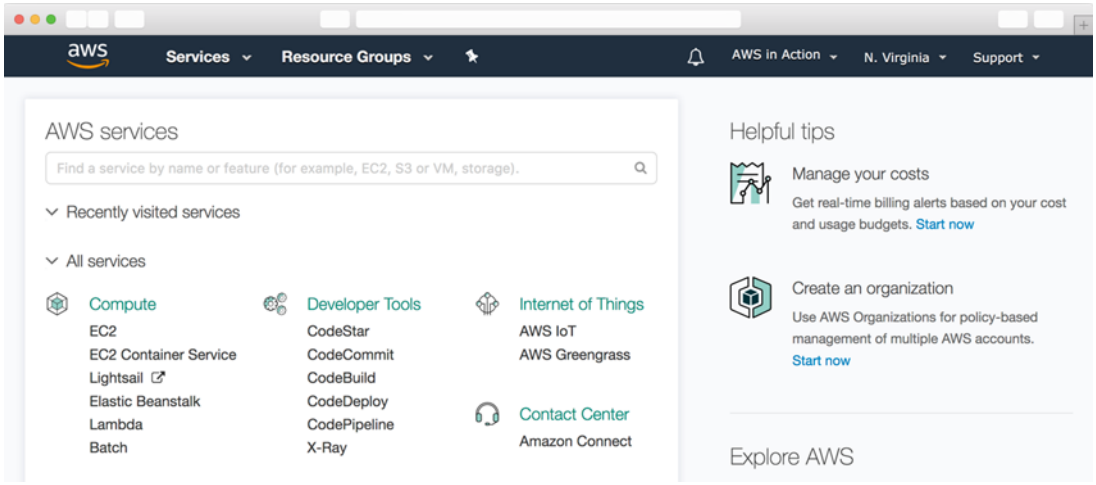


Figure 1.23 AWS Management Console

The most important part is the navigation bar at the top, shown in figure 1.24. It consists of seven sections:

- *AWS*—Start page of the Management Console, including an overview of all services.
- *Services*—Provides quick access to all AWS services.
- *Resource Groups*—Allows you to get an overview of all your AWS resources.
- *Custom section (Edit)*—Click the edit icon and drag-and-drop important services here to personalize the navigation bar.
- *Your name*—Lets you access billing information and your account, and also lets you sign out.
- *Your region*—Lets you choose your region. You will get to know about regions in section 3.5. You don't need to change anything here right now.
- *Support*—Gives you access to forums, documentation, and a ticket system.

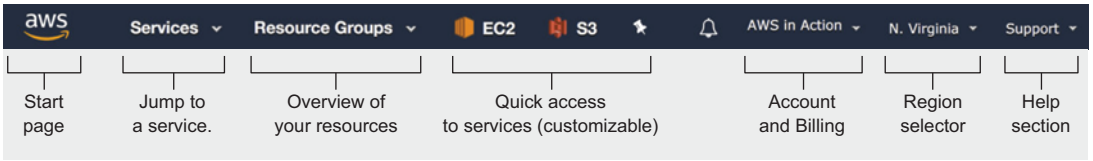


Figure 1.24 AWS Management Console navigation bar

Next, you'll create a key pair so you can connect to your virtual machines.

1.8.3 Creating a key pair

A *key pair* consists of a private key and a public key. The public key will be uploaded to AWS and injected into the virtual machine. The private key is yours; it's like your password, but much more secure. Protect your private key as if it were a password. It's your secret, so don't lose it—you can't retrieve it.

To access a Linux machine, you use the SSH protocol; you'll use a key pair for authentication instead of a password during login. When accessing a Windows machine via Remote Desktop Protocol (RDP), you'll need a key pair to decrypt the administrator password before you can log in.

> **Region US East (N. Virginia)**
>
> Amazon operates data centers in various geographic regions around the world. To simplify the examples, we're using the region US East (N. Virginia) within our book. You will also learn how to switch to another region to use resources in Asia Pacific (Sydney).
>
> Make sure you have selected the region US East (N. Virginia) before creating your key pair. Use the region selector in the navigation bar of the Management Console to change the region if needed.

The following steps will guide you to the dashboard of the EC2 service, which offers virtual machines, and where you can obtain a key pair:

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Click Services in the navigation bar and select EC2.
3. Your browser should now show the EC2 Dashboard.

The EC2 Dashboard, shown in figure 1.25, is split into three columns. The first column is the EC2 navigation bar; because EC2 is one of the oldest services, it has many features that you can access via the navigation bar. The second column gives you a brief overview of all your EC2 resources. The third column provides additional information.

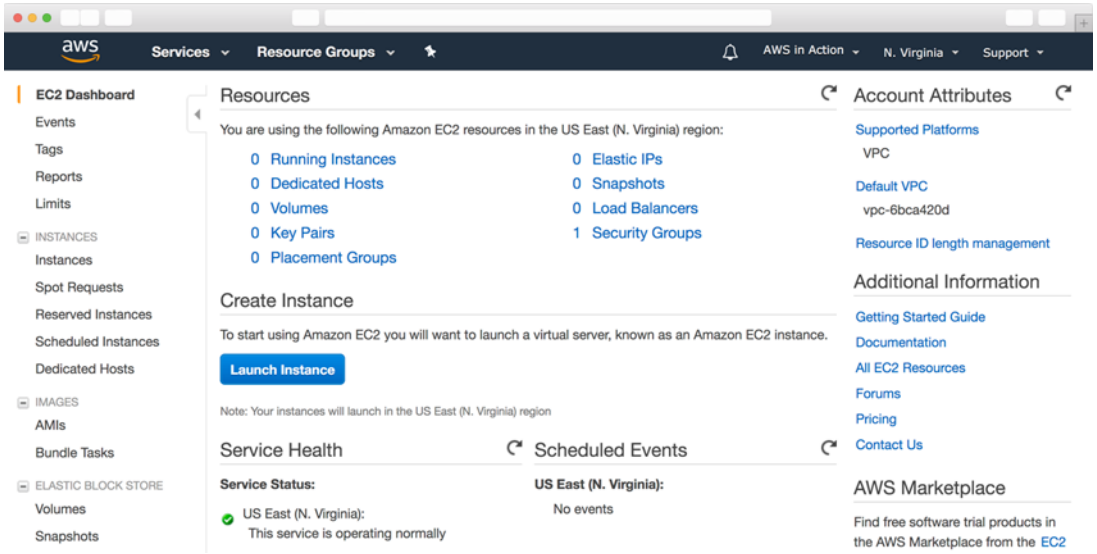


Figure 1.25 EC2 Management Console

Follow the steps in figure 1.26 to create a new key pair:

1. Click Key Pairs in the navigation bar under Network & Security.
2. Click the Create Key Pair button.
3. Name the Key Pair `mykey`. If you choose another name, you must replace the name in all the following examples during the whole book!

During the key-pair creation process, you download a file called mykey.pem. You must now prepare that key for future use. Depending on your operating system, you need to do things differently, so please read the section that fits your OS.

Figures with cueballs

In some figures, as in figure 1.26, you'll see numbered cueballs. They mark the order in which you need to click to follow the process being discussed in the text.

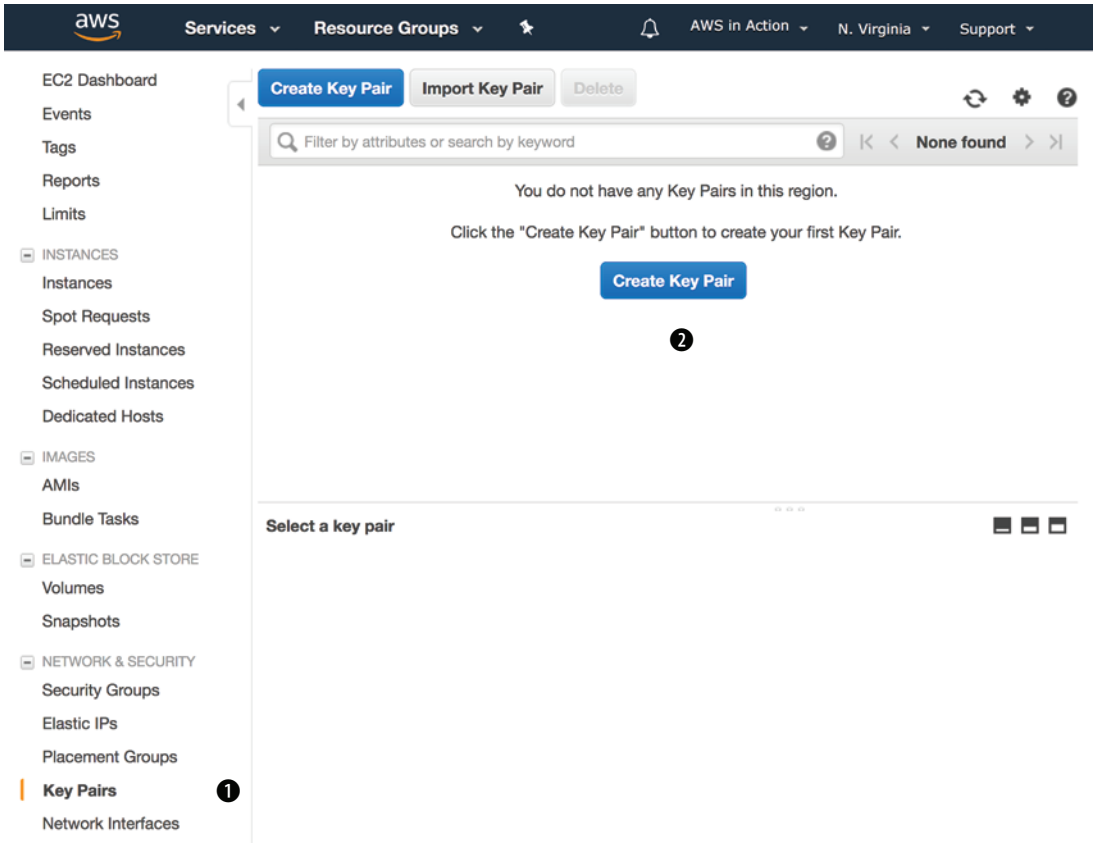


Figure 1.26 EC2 Management Console Key Pairs

Using your own key pair

It's also possible to upload the public key part from an existing key pair to AWS. Doing so has two advantages:

- You can reuse an existing key pair.
- You can be sure only you know the private key part of the key pair. If you use the Create Key Pair button, AWS knows (at least briefly) your private key.

We decided against that approach in this case because it's less convenient to implement in a book.

LINUX AND MACOS

The only thing you need to do is change the access rights of mykey.pem so that only you can read the file. To do so, run `chmod 400 mykey.pem` in the terminal. You'll learn about how to use your key when you need to log in to a virtual machine for the first time in this book.

WINDOWS

Windows doesn't ship an SSH client, so you need to download the PuTTY installer for Windows from http://mng.bz/A1bY and install PuTTY. PuTTY comes with a tool called PuTTYgen that can convert the mykey.pem file into a mykey.ppk file, which you'll need:

1. Run the PuTTYgen application. The screen shown in figure 1.27 opens. The most important steps are highlighted on the screen.
2. Select RSA (or SSH-2 RSA) under Type of Key to Generate.
3. Click Load.
4. Because PuTTYgen displays only *.ppk files, you need to switch the file extension of the File Name field to All Files.
5. Select the mykey.pem file, and click Open.
6. Confirm the dialog box.
7. Change Key Comment to `mykey`.
8. Click Save Private Key. Ignore the warning about saving the key without a passphrase.

Your .pem file has now been converted to the .ppk format needed by PuTTY. You'll learn about how to use your key when you need to log in to a virtual machine for the first time in this book.

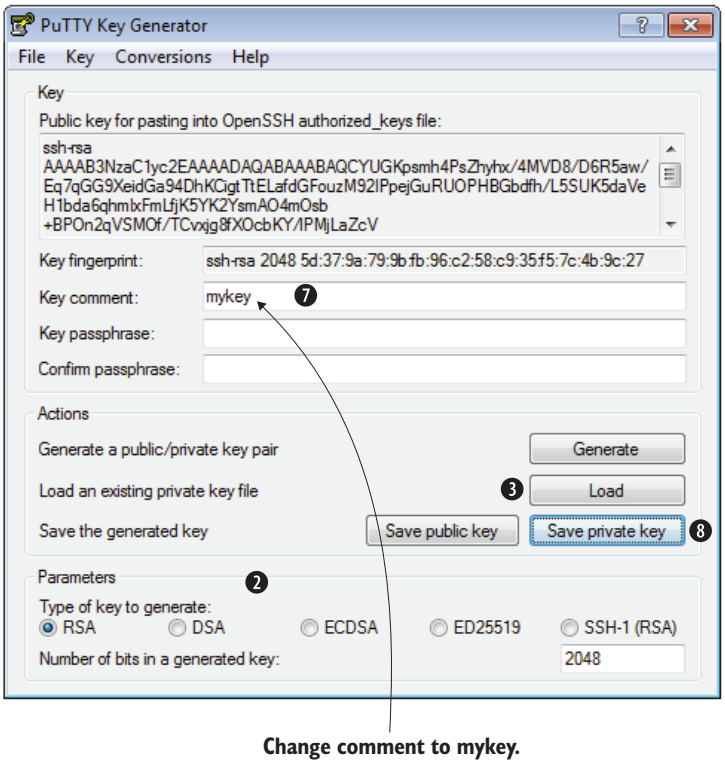


Figure 1.27 PuTTYgen allows you to convert the downloaded .pem file into the .ppk file format needed by PuTTY.

1.9 Create a billing alarm to keep track of your AWS bill

At first, the pay-per-use pricing model of AWS might feel unfamiliar to you, as it is not 100% foreseeable what your bill will look like at the end of the month. Most of the examples in this book are covered by the Free Tier, so AWS won't charge you anything. Exceptions are clearly marked. To provide you with the peace of mind needed to learn about AWS in a comfortable environment, you will create a billing alarm next. The billing alarm will notify you via email if your monthly AWS bill exceeds $5 USD so that you can react quickly.

First, you need to enable billing alarms within your AWS account. The step are illustrated in figure 1.28. The first step, of course, is to open the AWS Management Console at https://console.aws.amazon.com.

1. Click your Name in the main navigation bar on the top.
2. Select My Billing Dashboard from the pop-up menu.
3. Go to Preferences by using the sub navigation on the left side.
4. Select the Receive Billing Alerts check box.
5. Click Save preferences.

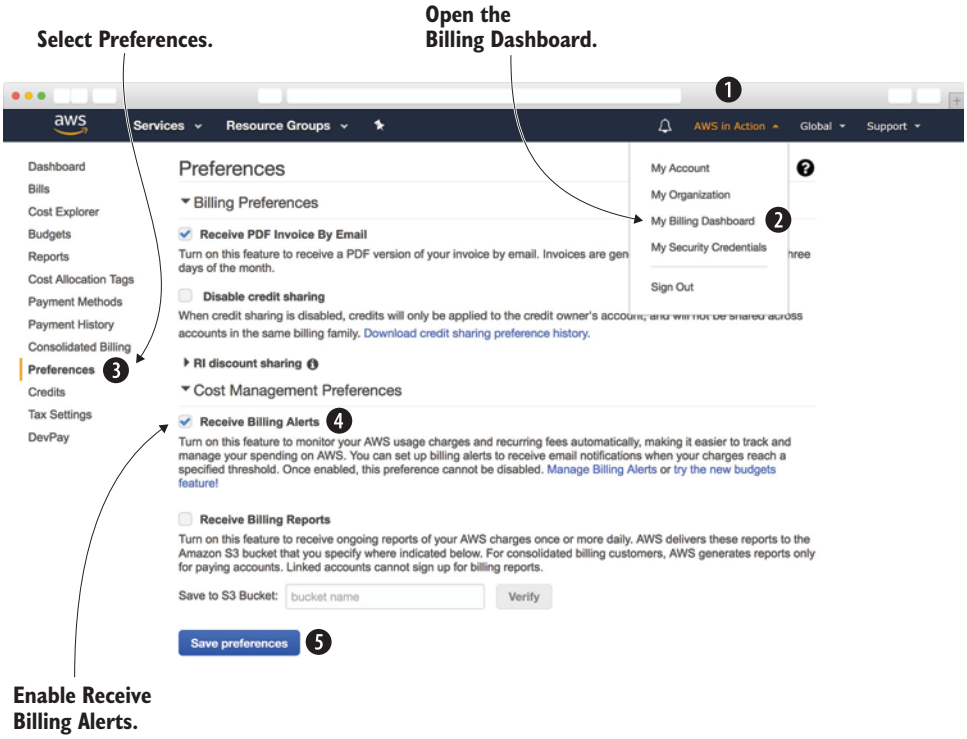


Figure 1.28 Creating a billing alarm (step 1 of 4)

You are now able to create a billing alarm. Here are the steps to do so:

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Open Services in the navigation bar and select CloudWatch.
3. Click the Create a billing alarm link as shown in figure 1.29.

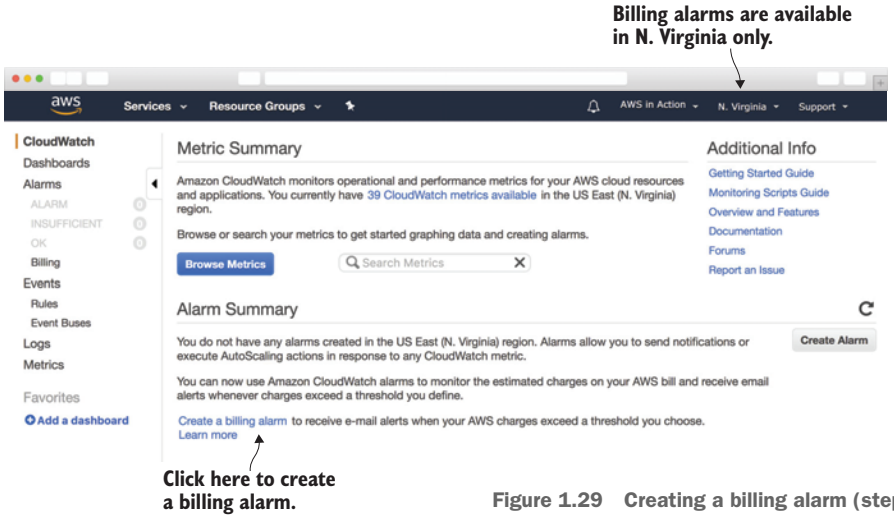


Figure 1.29 Creating a billing alarm (step 2 of 4)

Figure 1.30 shows the wizard that guides you through creating a billing alarm. Enter the threshold of total monthly charges for the billing alarm. We suggest $5 UDS, the equivalent of the price for a cup of coffee, as the threshold. Type in the email address where you want to receive notifications from your billing alarm in case your AWS bill exceeds the threshold. Click Create Alarm to create your billing alarm.

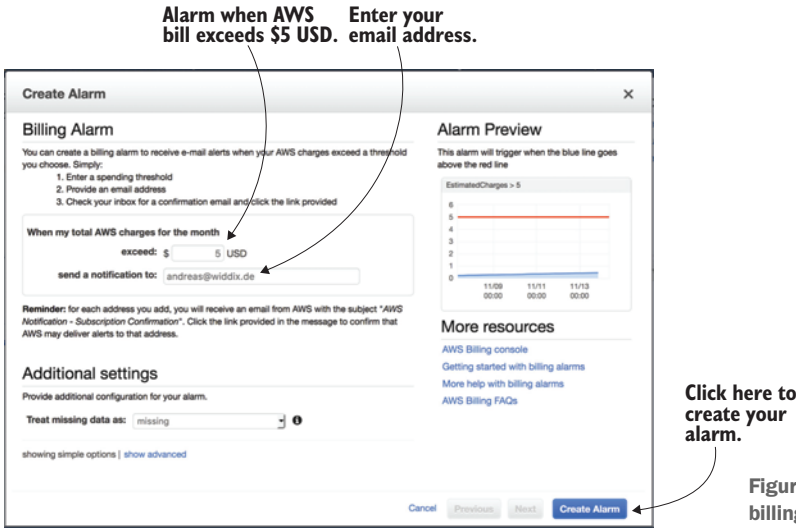


Figure 1.30 Creating a billing alarm (step 3 of 4)

Figure 1.31 Creating a billing alarm (step 4 of 4)

Open your inbox. An email from AWS containing a confirmation link appears. Click the confirmation link to complete the setup of your billing alarm. The dialog shown in figure 1.31 shows the status.

That's all. If your monthly AWS bill exceeds $5 USD for any reason, you will be notified immediately, allowing you to react before unwanted costs occur.

Summary

- Amazon Web Services (AWS) is a platform of web services for computing, storing, and networking that work well together.
- Cost savings aren't the only benefit of using AWS. You'll also profit from an innovative and fast-growing platform with flexible capacity, fault-tolerant services, and a worldwide infrastructure.
- Any use case can be implemented on AWS, whether it's a widely used web application or a specialized enterprise application with an advanced networking setup.
- You can interact with AWS in many different ways. You can control the different services by using the web-based GUI, use code to manage AWS programmatically from the command line or SDKs, or use blueprints to set up, modify, or delete your infrastructure on AWS.
- Pay-per-use is the pricing model for AWS services. Computing power, storage, and networking services are billed similarly to electricity.
- Creating an AWS account is easy. Now you know how to set up a key pair so you can log in to virtual machines for later use.
- Creating a billing alarm allows you to keep track of your AWS bill and get notified whenever you exceed the Free Tier.

2 A simple example: WordPress in five minutes

This chapter covers

- Creating a blogging infrastructure
- Analyzing costs of a blogging infrastructure
- Exploring a blogging infrastructure
- Shutting down a blogging infrastructure

Having looked at why AWS is such a great choice to run web applications in the cloud, in this chapter, you'll evaluate migrating a simple web application to AWS by setting up a sample cloud infrastructure within five minutes.

> **NOTE** The example in this chapter is totally covered by the Free Tier (see section 1.4.1 for details). As long as you don't run this example longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

Imagine you work for a mid-sized company that runs a blog to attract new software and operations engineers. WordPress is used as the content management system.

Around 1,000 people visit the blog daily. You are paying $150 USD per month for the on-premises infrastructure. This seems expensive to you, particularly because at the moment the blog is suffering from several outages per month.

To leave a good impression on potential candidates, the infrastructure should be highly available, which is defined as an uptime of 99.99%. Therefore, you are evaluating new options to operate WordPress reliably. AWS seems to be a good fit. As a proof-of-concept, you want to evaluate whether a migration is possible. To do so, you need to do the following:

- Set up a highly available infrastructure for WordPress.
- Estimate monthly costs of the infrastructure.
- Come to a decision and delete the infrastructure afterward.

WordPress is written in PHP and uses a MySQL database to store data. Apache is used as the web server to serve the pages. With this information in mind, it's time to map your requirements to AWS services.

2.1 Creating your infrastructure

You'll use five different AWS services to copy the old infrastructure to AWS:

- *Elastic Load Balancing (ELB)*—AWS offers a load balancer as a service. The load balancer distributes traffic to a bunch of virtual machines, and is highly available by default. Requests are routed to virtual machines as long as their health check succeeds. You'll use the Application Load Balancer (ALB) which operates on Layer 7 (HTTP and HTTPS).
- *Elastic Compute Cloud (EC2)*—The EC2 service provides virtual machines. You'll use a Linux machine with an optimized distribution called Amazon Linux to install Apache, PHP, and WordPress. You aren't limited to Amazon Linux; you could also choose Ubuntu, Debian, Red Hat, or Windows. Virtual machines can fail, so you need at least two of them. The load balancer will distribute the traffic between them. In case a virtual machine fails, the load balancer will stop sending traffic to the failed VM, and the remaining VM will need to handle all requests until the failed VM is replaced.
- *Relational Database Service (RDS) for MySQL*—WordPress relies on the popular MySQL database. AWS provides MySQL with its RDS. You choose the database size (storage, CPU, RAM), and RDS takes over operating tasks like creating backups and installing patches and updates. RDS can also provide a highly available MySQL database by replication.
- *Elastic File System (EFS)*—WordPress itself consists of PHP and other application files. User uploads, for example images added to an article, are stored as files as well. By using a network file system, your virtual machines can access these files. EFS provides a scalable, highly available, and durable network filesystem using the NFSv4.1 protocol.

- *Security groups*— Control incoming and outgoing traffic to your virtual machine, your database, or your load balancer with a firewall. For example, use a security group allowing incoming HTTP traffic from the internet to port 80 of the load balancer. Or restrict network access to your database on port 3306 to the virtual machines running your web servers.

Figure 2.1 shows all the parts of the infrastructure in action. Sounds like a lot of stuff to set up, so let's get started!

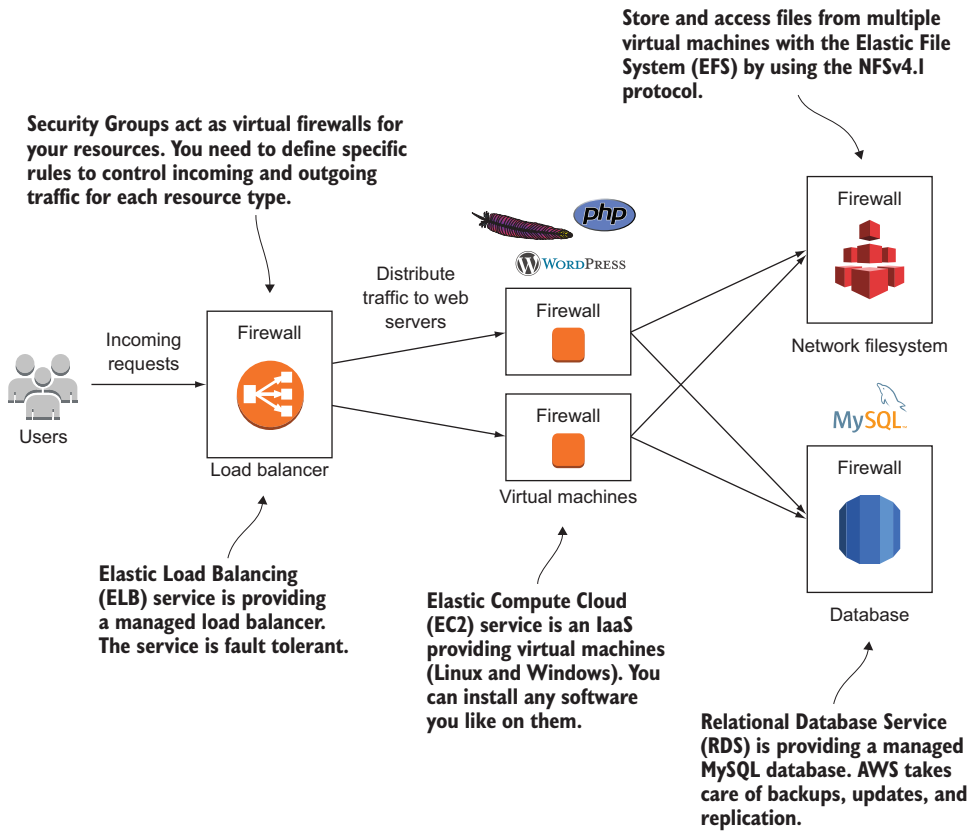


Figure 2.1 The company's blogging infrastructure consists of two load-balanced web servers running WordPress, a network filesystem, and a MySQL database server.

If you expect pages of instructions, you'll be happy to know that you can create all that with just a few clicks. Doing so is possible by using a service called AWS CloudFormation that you will learn about in detail in chapter 4. AWS CloudFormation will do all of the following automatically in the background:

1. Create a load balancer (ELB).
2. Create a MySQL database (RDS).

3. Create a network filesystem (EFS).
4. Create and attach firewall rules (security groups).
5. Create two virtual machines running web servers:
   a. Create two virtual machines (EC2).
   b. Mount the network filesystem.
   c. Install Apache and PHP.
   d. Download and extract the 4.8 release of WordPress.
   e. Configure WordPress to use the created MySQL database (RDS).
   f. Start the Apache web server.

To create the infrastructure for your proof-of-concept, open the AWS Management Console at https://console.aws.amazon.com. Click Services in the navigation bar, and select the CloudFormation service. You can use the search function to find CloudFormation more easily. You'll see a page like the one shown in figure 2.2.

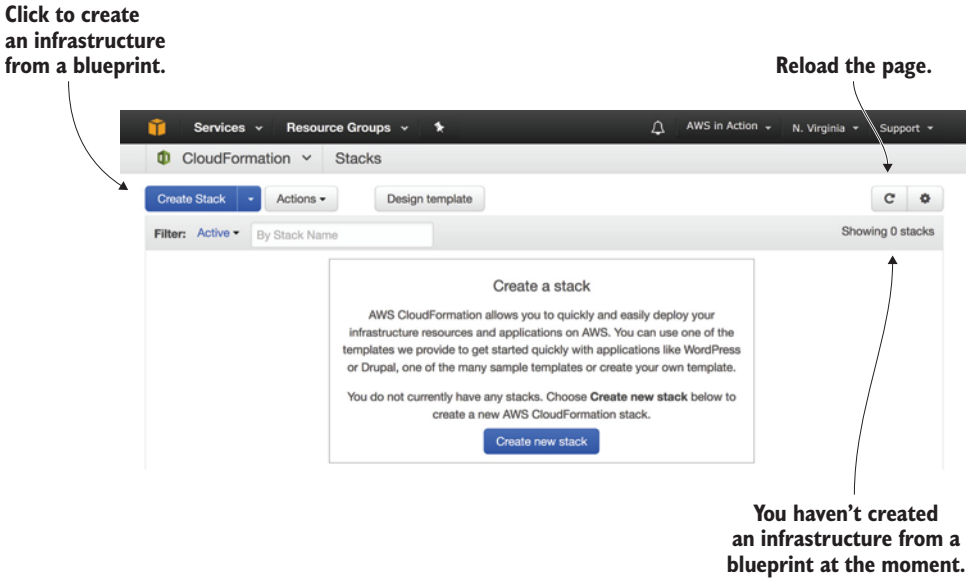


Figure 2.2 Overview of CloudFormation

DEFAULT REGION FOR EXAMPLES All examples in this book use N. Virginia (also called us-east-1) as the default region. Exceptions are indicated. Please make sure you switch to the region N. Virginia before starting to work on an example. When using the AWS Management Console, you can check and switch the region on the right side of the main navigation bar at the top.

Click Create Stack to start the four-step wizard, as shown in figure 2.3.

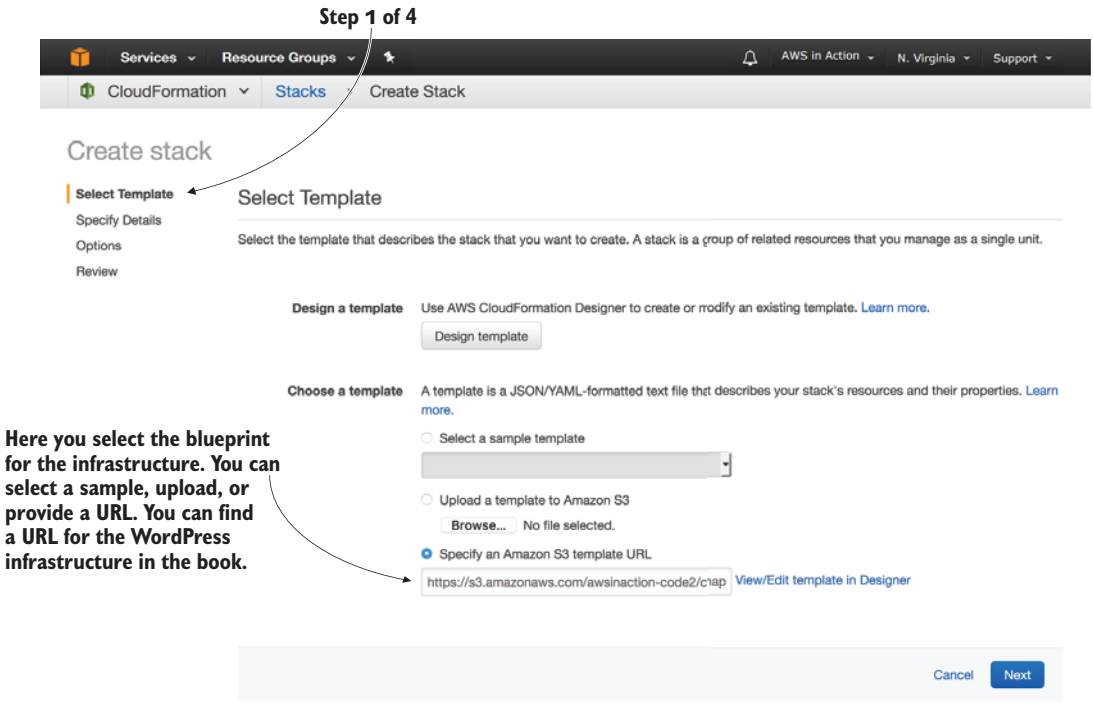


Figure 2.3 Creating the stack for the proof-of-concept: step 1 of 4

Choose Specify an Amazon S3 template URL and enter `https://s3.amazonaws.com/awsinaction-code2/chapter02/template.yaml` to use the template prepared for this chapter. Proceed with the next step of the wizard.

Specify `wordpress` as the Stack name and set the `KeyName` to `mykey` in the Parameters section as shown in figure 2.4.

The next step is to specify tags for your infrastructure, as illustrated by figure 2.5. A *tag* consists of a key and a value, and can be used to add metadata to all parts of your infrastructure. You can use tags to differentiate between testing and production resources, add a cost center to easily track costs in your organization, or mark resources that belong to a certain application if you host multiple applications in the same AWS account.

Figure 2.5 shows how to configure the tag. In this example, you'll use a tag to mark all resources that belong to the `wordpress` system. This will help you to easily find all the parts of your infrastructure later. Use a custom tag consisting of the key `system` and the value `wordpress`. Afterward, press the Next button to proceed to the next step. You can define your own tags as long as the key name is shorter than 128 characters and the value has fewer than 256 characters.

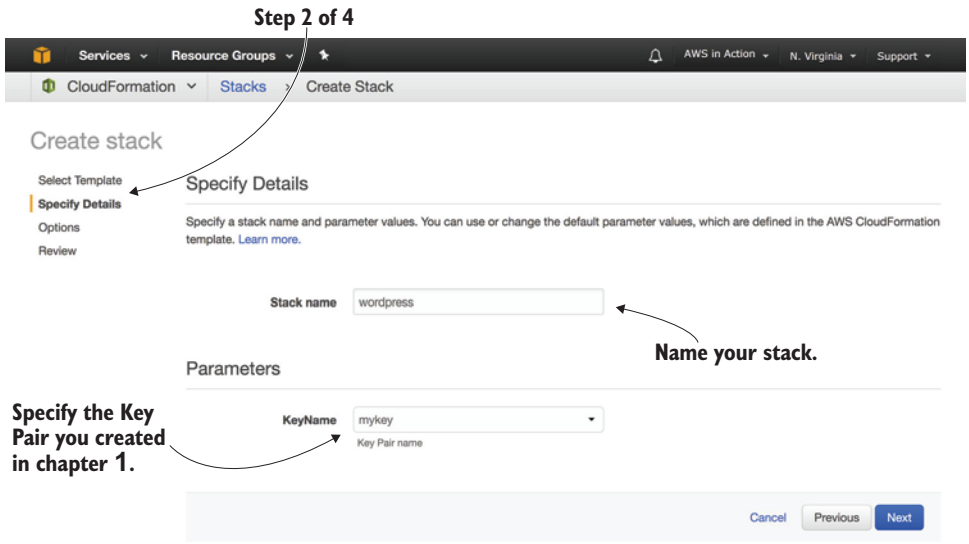


Figure 2.4 Creating the stack for the proof-of-concept: step 2 of 4

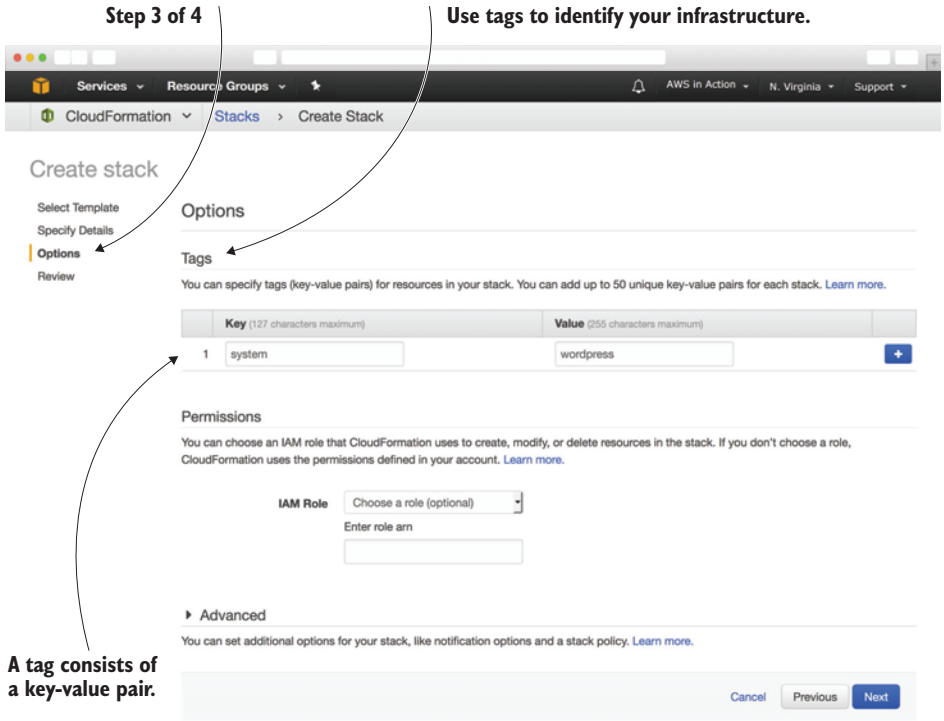


Figure 2.5 Creating the stack for the proof-of-concept: step 3 of 4

Additional CloudFormation Stack options

It is possible to define specific permissions used to manage resources, as well as to set up notifications and other advanced options. You won't need these options for 99% of the use cases, so we don't cover them in our book. Have a look at the CloudFormation User Guide (http://mng.bz/njoZ) if you're interested in the details.

Figure 2.6 shows the confirmation page. Please click on the Estimate cost link to open a cost estimation of your cloud infrastructure in a new browser tab. Don't worry, the example is covered by the Free Tier and you'll learn about all the details of the cost estimation in section 2.3. Switch back to the original browser tab, and click Create.

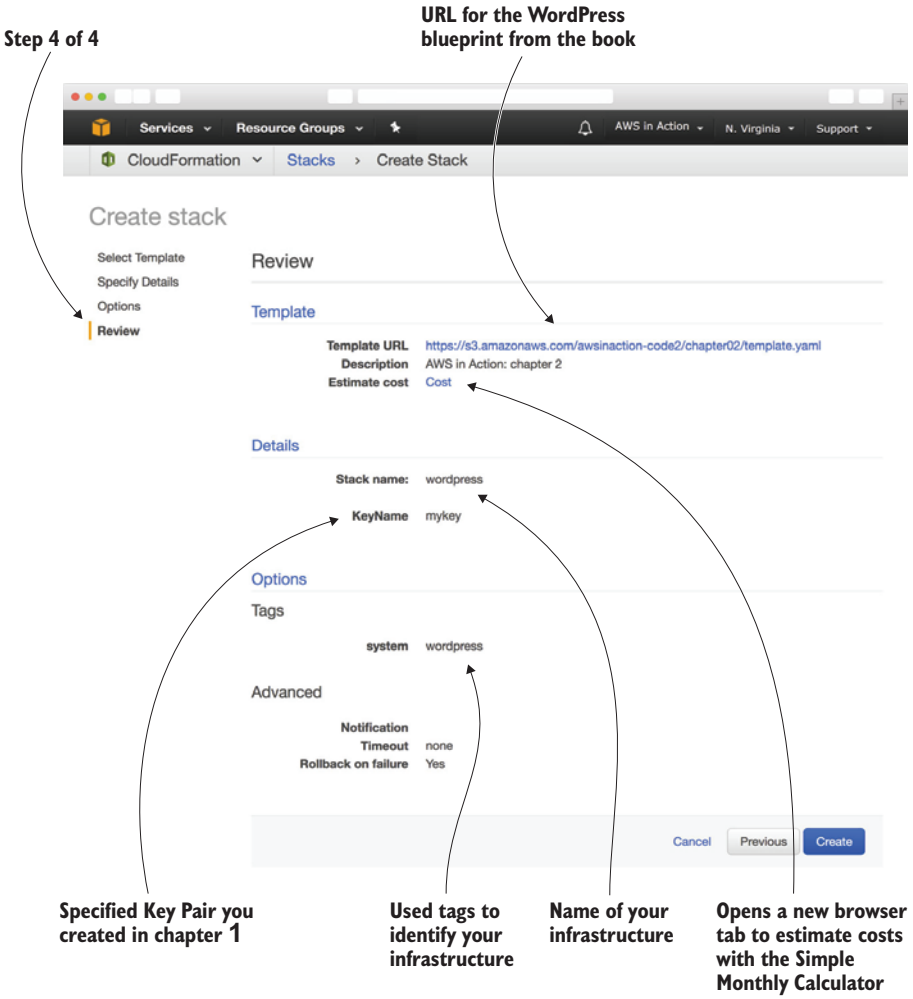


Figure 2.6 Creating the stack for the proof-of-concept: step 4 of 4

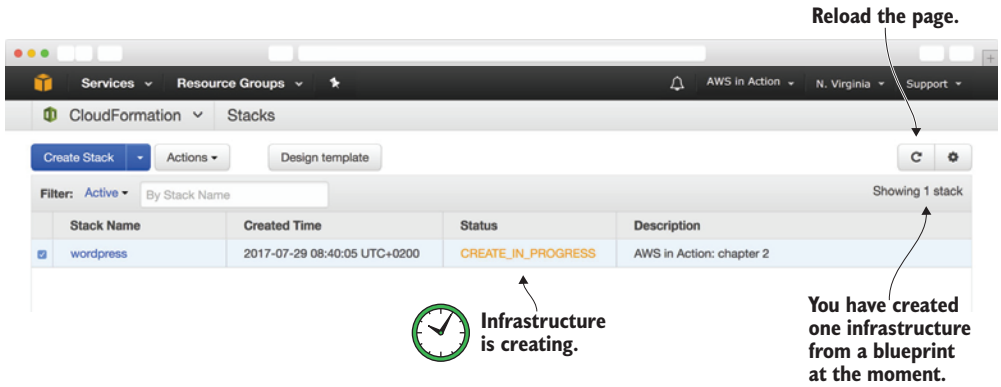


Figure 2.7 CloudFormation is creating the resources needed for WordPress.

Your infrastructure will now be created. Figure 2.7 shows that `wordpress` is in state `CREATE_IN_PROGRESS`. It's a good time to take a break; come back in five minutes, and you'll be surprised.

After all the needed resources have been created, the status will change to `CREATE_COMPLETE`. Be patient and hit the refresh icon from time to time if your status still shows as `CREATE_IN_PROGRESS`.

Select the check box at the beginning of the row containing your wordpress stack. Switch to the Outputs tab, as shown in figure 2.8. There you'll find the URL to your WordPress installation; click the link to open it in your browser.

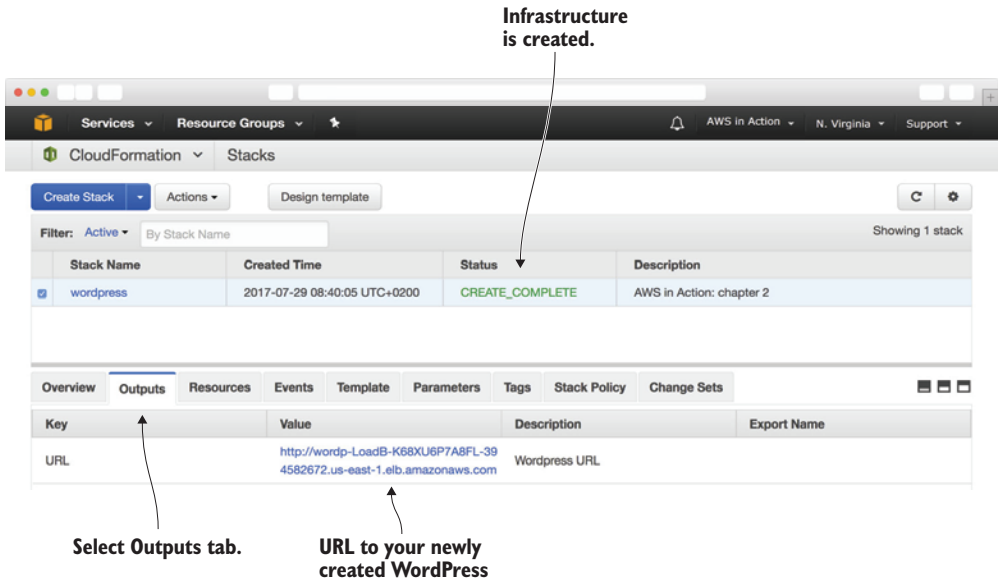


Figure 2.8 Blogging infrastructure has been created successfully.

You may ask yourself, how does this work? The answer is *automation*.

> **Automation references**
>
> One of the key concepts of AWS is automation. You can automate everything. In the background, your blogging infrastructure was created based on a blueprint. You'll learn more about blueprints and the concept of programming your infrastructure in chapter 4. You'll learn to automate the installation of software in chapter 5.

You'll explore the blogging infrastructure in the next section to get a better understanding of the services you're using.

2.2 Exploring your infrastructure

Now that you've created your blogging infrastructure, let's take a closer look at it. Your infrastructure consists of the following:

- Web servers running on virtual machines
- Load balancer
- MySQL database
- Network filesystem

You'll use the Management Console's resource groups feature to get an overview.

2.2.1 Resource groups

A *resource group* is a collection of AWS resources. *Resource* in AWS is an abstract term for something like an virtual machine, a security group, or a database. Resources can be tagged with key-value pairs, and resource groups specify which tags are needed for a resource to belong to the group. Furthermore, a resource group specifies the region(s) the resource must reside in. You can use resource groups for grouping resources if you run multiple systems in the same AWS account.

Remember that you tagged the blogging infrastructure with the key `system` and the value `wordpress`. From now on, we'll use this notation for key-value pairs: `(system:wordpress)`. You'll use that tag to create a resource group for your WordPress infrastructure. Click Create a Resource Group as shown in figure 2.9.

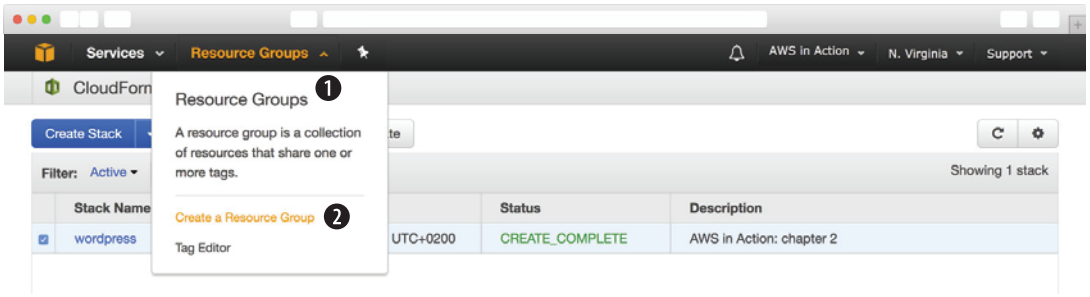


Figure 2.9 Open resource groups

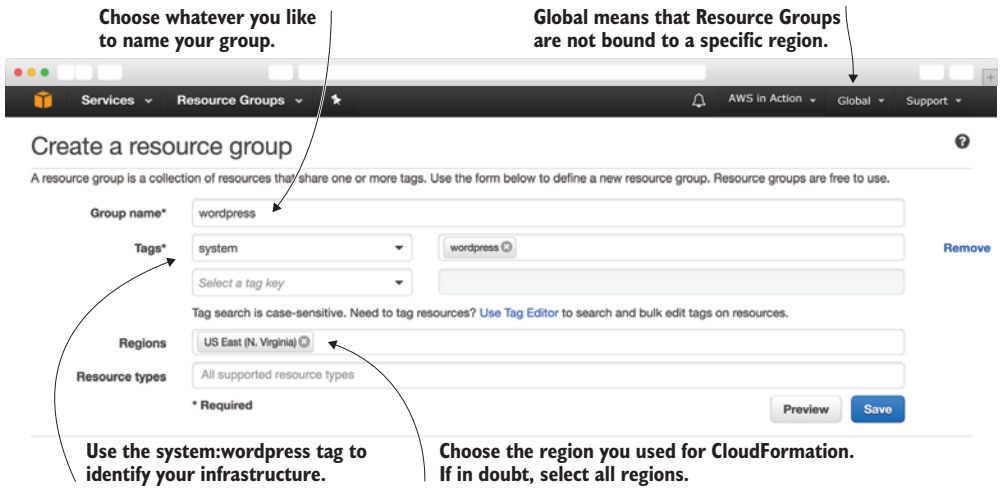


Figure 2.10 Creating a resource group for your blogging infrastructure

You'll now create a resource group as illustrated in 2.10:

1. Set Group Name to `wordpress` or whatever you like.
2. Add the tag `system` with the value `wordpress`.
3. Select the region you're in, which is probably N. Virginia. If you don't know the region you're in, select All Regions.
4. Click Save.

2.2.2 Virtual machines

Now you'll see the screen shown in figure 2.11. Select Instances under EC2 on the left to see your virtual machines. By clicking the arrow icon in the Go column, you can easily jump to the details of a single virtual machine.

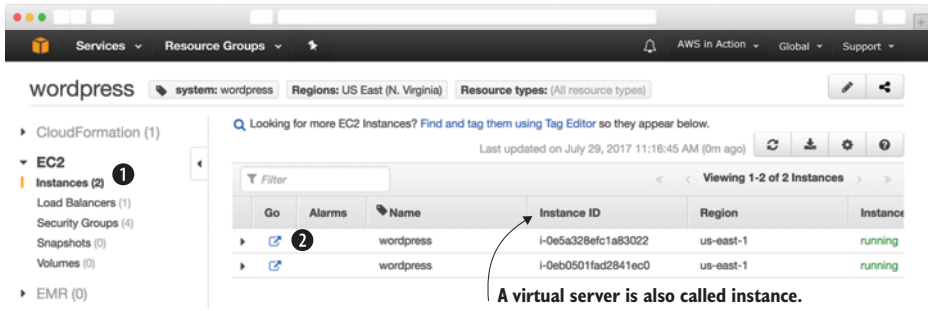


Figure 2.11 Blogging infrastructure virtual machines via resource groups

You're now looking at the details of your virtual machine, also called an EC2 instance. Figure 2.12 shows an extract of what you see. The interesting details are as follows:

- *Instance type*—Tells you about how powerful your EC2 instance is. You'll learn more about instance types in chapter 3.
- *IPv4 Public IP*—The IP address that is reachable over the internet. You can use that IP address to connect to the virtual machine via SSH.
- *Security groups*—If you click View Rules, you'll see the active firewall rules, like the one that enables port 22 from all sources (0.0.0.0/0)
- *AMI ID*—Remember that you used the Amazon Linux OS. If you click the AMI ID, you'll see the version number of the OS, among other things.

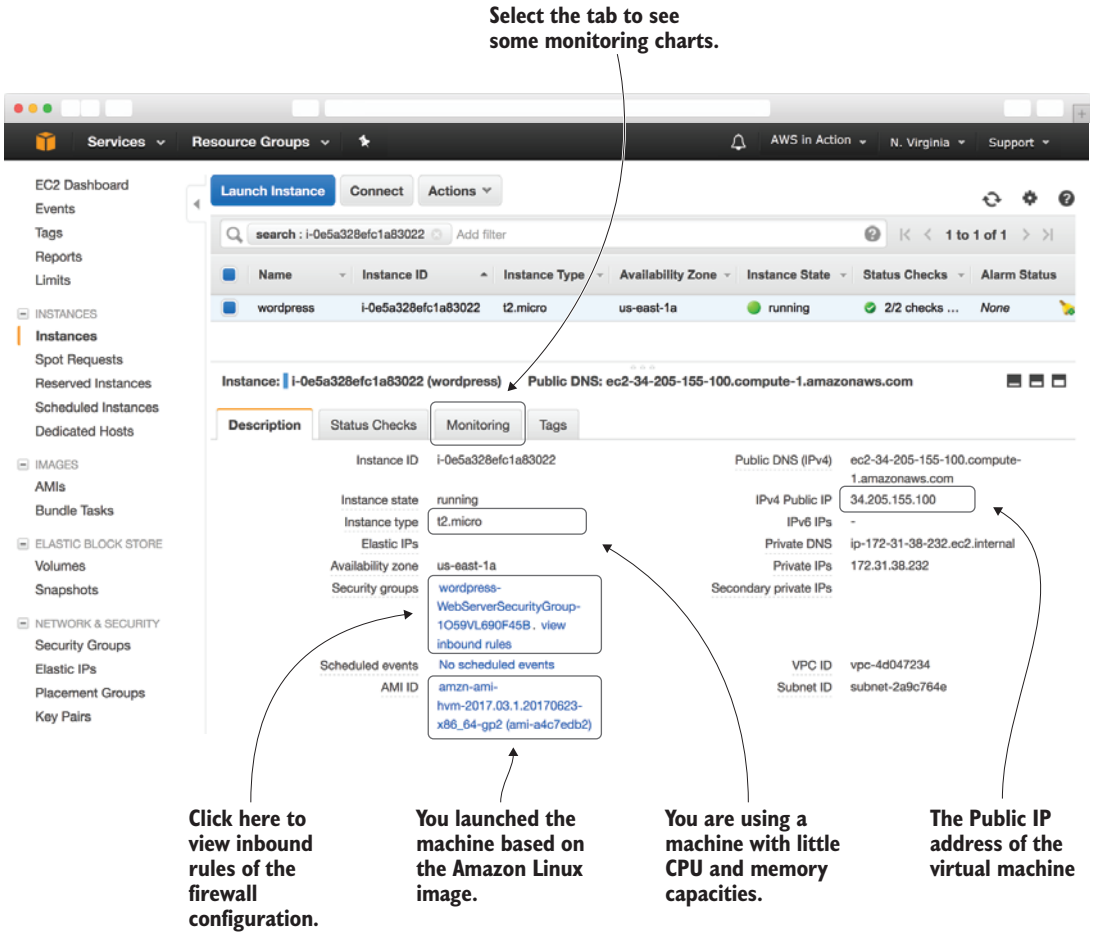


Figure 2.12 Details of web servers running the blogging infrastructure

Select the Monitoring tab to see how your virtual machine is utilized. This tab is essential if you really want to know how your infrastructure is doing. AWS collects some metrics and shows them here. For example, if the CPU is utilized more than 80%, it might be a good time to add another virtual machine to prevent increasing response times. You will learn more about monitoring virtual machines in section 3.2.

2.2.3 Load balancer

AWS released a new load balancer type, called Application Load Balancer, in August 2016. Unfortunately, our resource group does not list Application Load Balancers yet. Therefore, click Load Balancers in the sub navigation of the EC2 service as shown in figure 2.13.

Select your load balancer from the list to show more details. Your internet-facing load balancer is accessible from the internet via an automatically generated DNS name.

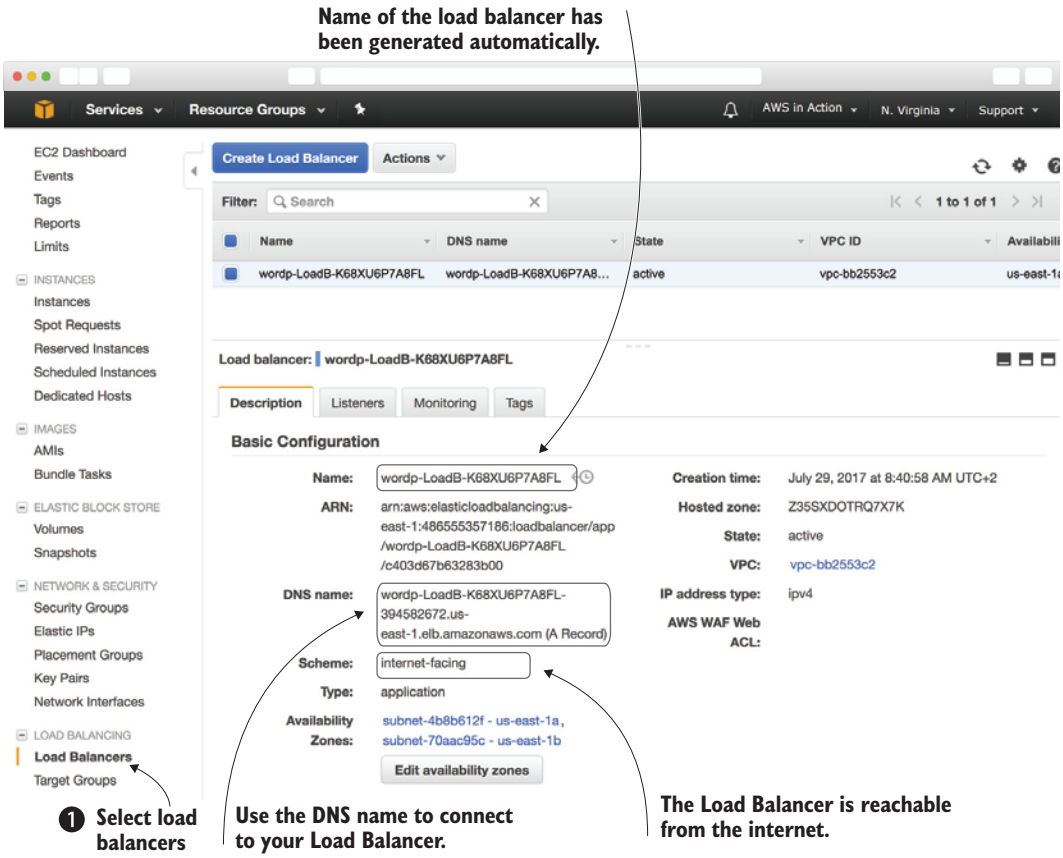


Figure 2.13 Get details about your load balancer.

The load balancer forwards incoming requests to one of your virtual machines. A target group is used to define the targets for a load balancer. You'll find your target group after switching to Target Groups through the sub navigation of the EC2 service as shown in figure 2.14.

The load balancer performs health checks to ensure requests are routed to healthy targets only. Two virtual machines are listed as targets for the target group. As you can see in the figure, the status of both virtual machines is healthy.

As before, there is a Monitoring tab where you can find interesting metrics that you should watch in production. If the traffic pattern changes suddenly, this indicates a potential problem with your system. You'll also find metrics indicating the number of HTTP errors, which will help you to monitor and debug your system.

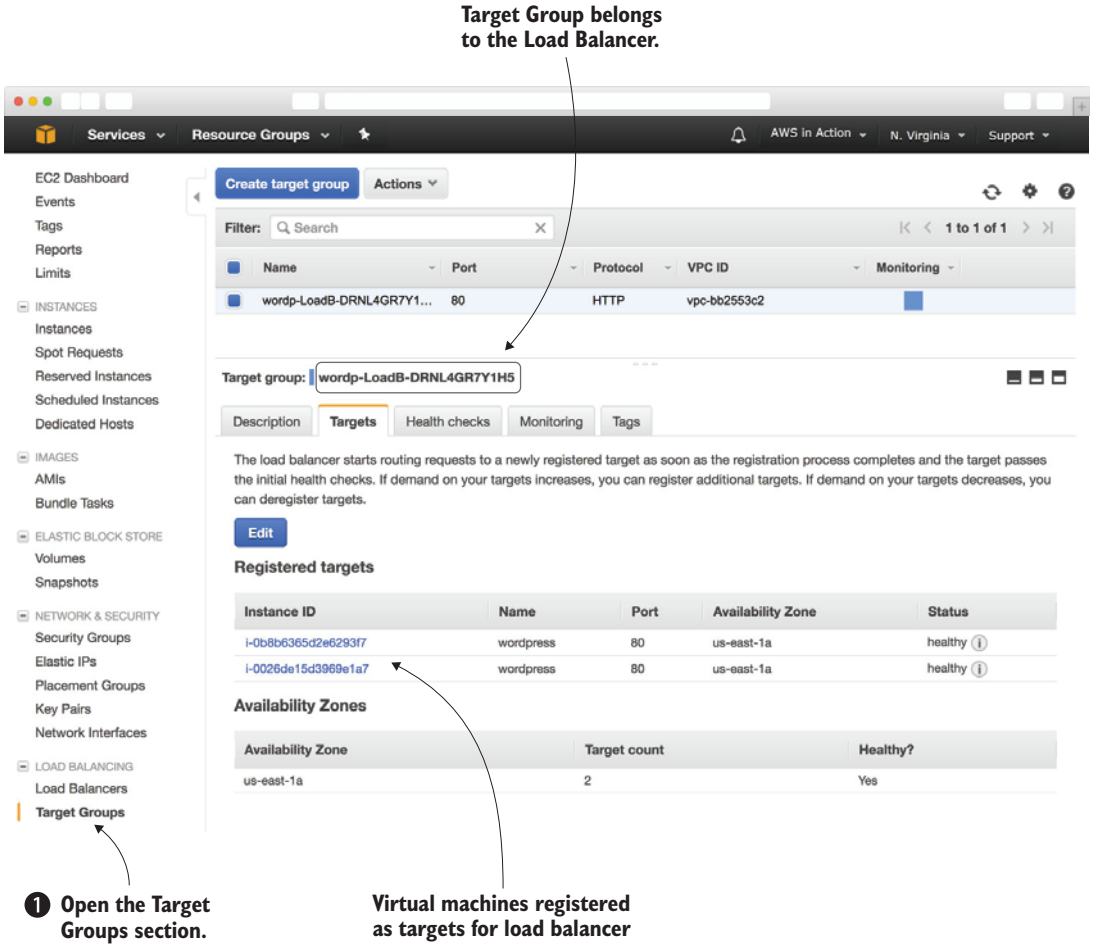


Figure 2.14 Details of target group belonging to the load balancer

2.2.4 MySQL database

The MySQL database is an important part of your infrastucture; you'll look at it next. Go back to the resource group named `wordpress`. Select DB Instances under RDS on the left. By clicking the arrow icon in the Go column, as shown in figure 2.15, you can easily jump to the details of the database.

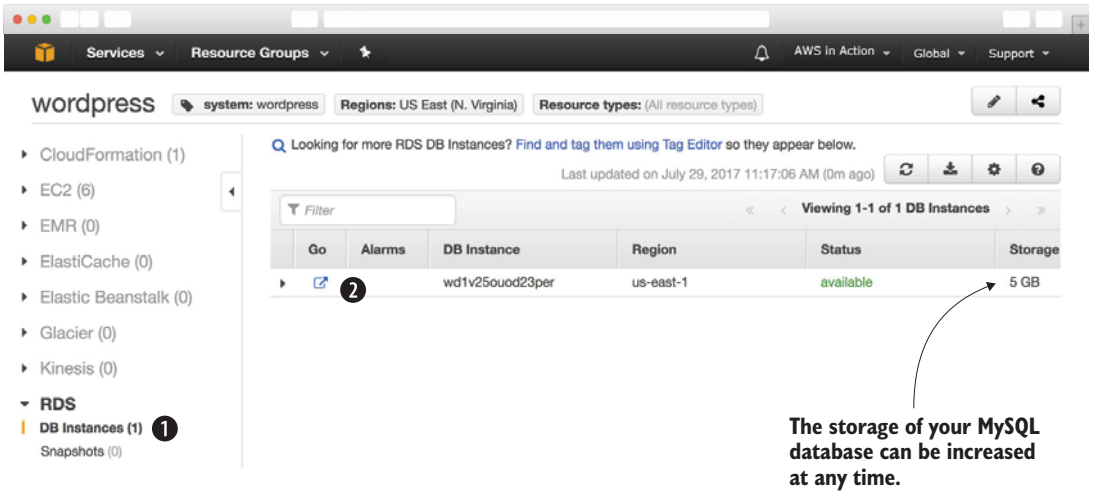


Figure 2.15 Blogging infrastructure MySQL via resource groups

The details of your MySQL database are shown in figure 2.16. The RDS offers SQL databases as managed services, complete with backups, patch management, and high availability. As shown in figure 2.16 automated backups are disabled, as they are not needed for our proof-of-concept without any critical data. You can also find the maintenance window used by AWS to apply patches automatically in the Details section.

WordPress requires a MySQL database, so you have launched a database instance with the MySQL engine as noted in figure 2.16. Your blog receives a low amount of traffic, so the database doesn't need to be very powerful. A small instance class with a single virtual CPU and 1 GB memory is sufficient. Instead of using SSD storage, you are using magnetic disks, which is cheaper and sufficient for a web application with around 1,000 visitors per day.

As you'll see in chapter 9, other database engines, such as PostgreSQL or Oracle Database, are available as well as more powerful instance classes, offering up to 32 cores with 244 GB memory.

Common web applications use a database to store and query data. That is true for WordPress as well. The Content Management System (CMS) stores blog posts, comments, and more within a MySQL database.

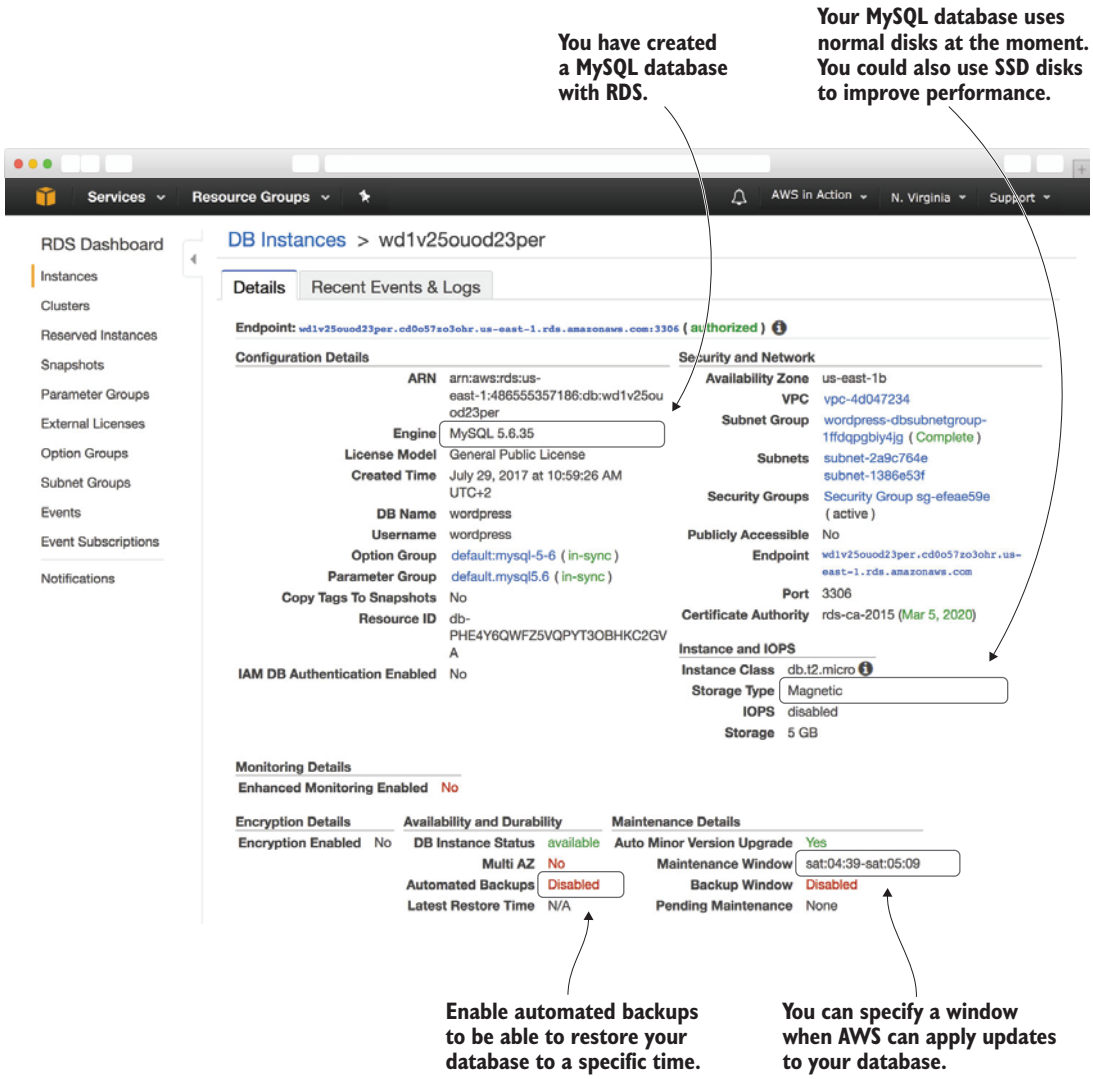


Figure 2.16 Details of the MySQL database storing data for the blogging infrastructure

But WordPress also stores data outside the database on disk. For example, if an author uploads an image for their blog post, the file is stored on disk. The same is true when you are installing plug-ins and themes as an administrator.

2.2.5 Network filesystem

The EFS is used to store files and access them from multiple virtual machines. EFS is a storage service accessible through the NFS protocol. To keep things simple, all files that belong to WordPress are stored on EFS so they can be accessed from all virtual machines. This includes PHP, HTML, CSS, and PNG.

EFS is not available from your resource group, unfortunately. Select EFS from the service menu to get more information about your NFS, as shown in figure 2.17. You can find the name, DNS name, and the mount targets of the filesystem in the Details section.

To mount the Elastic File System from a virtual machine, mount targets are needed. You should use two mount targets for fault tolerance. The network filesystem is accessible using a DNS name for the virtual machines.

Now it's time to evaluate costs. You'll analyze the costs of your blogging infrastructure in the next section.

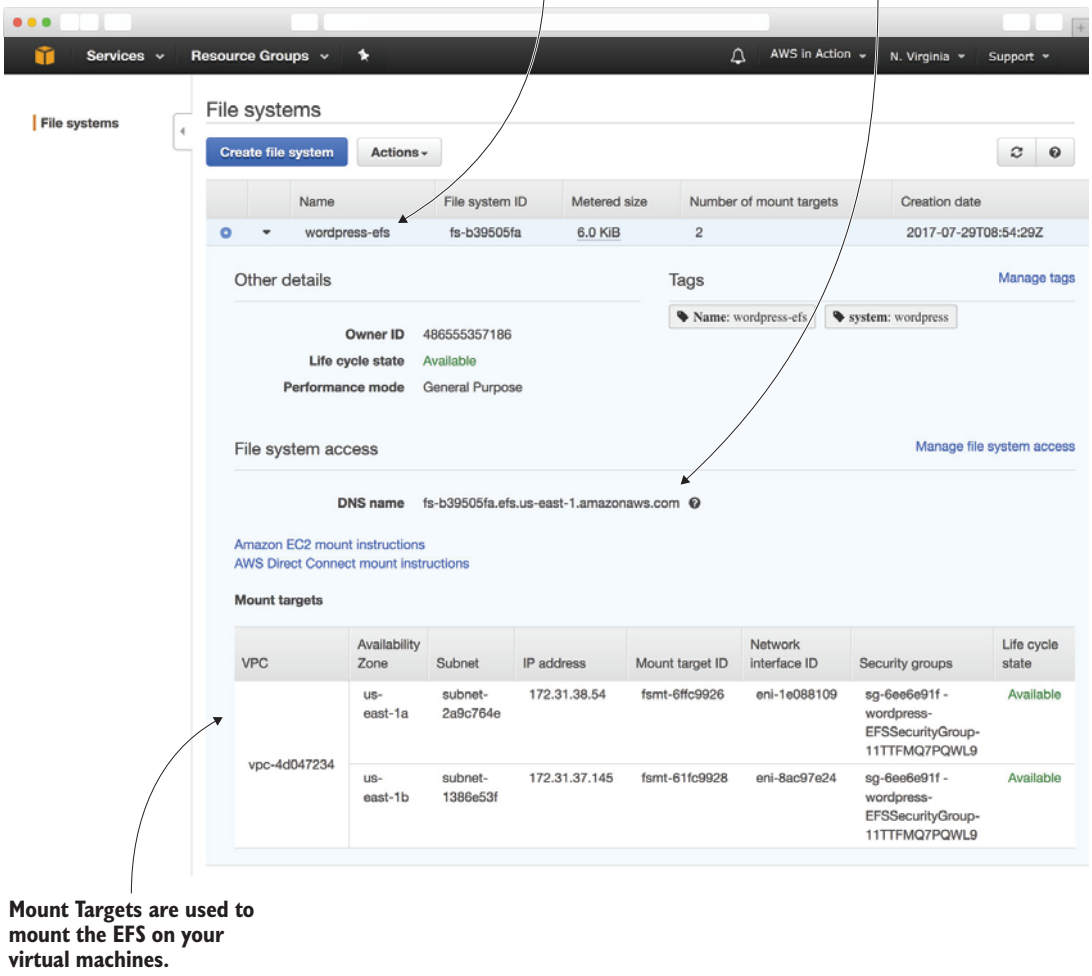


Figure 2.17 NFS used to store the WordPress application and user uploads

2.3 How much does it cost?

Part of evaluating AWS is estimating costs. To analyze the cost of your blogging infrastructure, you'll use the AWS Simple Monthly Calculator. Remember that you clicked the Cost link in the previous section to open a new browser tab. Switch to that tab, and you'll see a screen like that in figure 2.18. If you closed the tab, go to http://mng.bz/x6A0 instead. Click Estimate of your Monthly Bill, and expand the rows marked Amazon EC2 Service and Amazon RDS Service.

In this example, your infrastructure will cost around $35 USD per month. Prices for some services vary per region. That's why the estimation of your monthly bill may be different if you choose another region than N. Virginia (us-east-1).

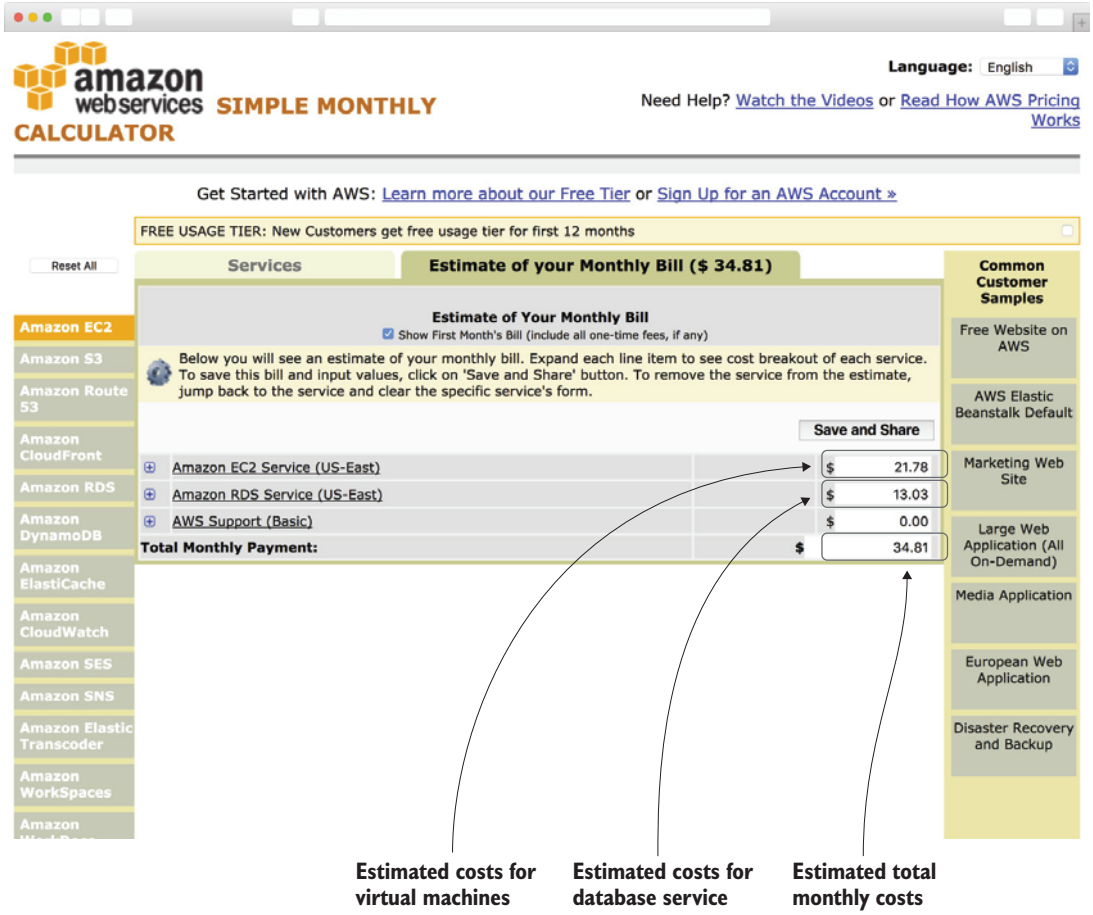


Figure 2.18 Blogging infrastructure cost calculation

Unfortunately the somewhat-new Application Load Balancer is not included in the estimation yet. The estimation is also missing some other details. Table 2.1 shows a more accurate cost calculation.

Table 2.1 More detailed cost calculation for blogging infrastructure

AWS service	Infrastructure	Pricing	Monthly cost
EC2	Virtual machines	2 * 732.5 hours * $0.012 (t2.micro) 2 * $2.10 (detailed monitoring)	$21.78
EC2	Storage	2 * 8 GB * $0.10 per month	$1.60
Application Load Balancer	Load balancer	732.5 hours * $0.0225 (load balancer hour) 732.5 hours * $0.008 (load balancer capacity unit)	$22.34
Application Load Balancer	Outgoing traffic	1 GB * $0.00 (first GB) 99 GB * $0.09 (up to 10 TB)	$8.91
RDS	MySQL database instance	732.5 hours * $0.017	$12.45
RDS	Storage	5 GB * $0.115	$0.58
EFS	Storage	5 GB * $0.3	$1.50
			$69.16

Keep in mind that this is only an estimate. You're billed based on actual use at the end of the month. Everything is on-demand and usually billed by seconds or gigabyte of usage. But what factors might influence how much you actually use this infrastructure?

- *Traffic processed by the load balancer*—Expect costs to go down in December and in the summer when people are on vacation and not looking at your blog.
- *Storage needed for the database*—If your company increases the amount of content in your blog, the database will grow, so the cost of storage will increase.
- *Storage needed on the NFS*—User uploads, plug-ins, and themes increase the amount of storage needed on the NFS, which will also increase costs.
- *Number of virtual machines needed*—Virtual machines are billed by seconds of usage. If two virtual machines aren't enough to handle all the traffic during the day, you may need a third machine. In that case, you'll consume more seconds of virtual machines.

Estimating the cost of your infrastructure is a complicated task, but that is also true if your infrastructure doesn't run in AWS. The benefit of using AWS is that it's flexible. If your estimated number of virtual machines is too high, you can get rid of a machine and stop paying for it. You will learn more about the pricing model of the different AWS services during the course of this book.

You have completed the proof-of-concept for migrating your company's blog to AWS. It's time to shut down the infrastructure and complete your migration evaluation.

2.4 Deleting your infrastructure

Your evaluation has confirmed that you can migrate the infrastructure needed for the company's blog to AWS from a technical standpoint. You have estimated that a load balancer, virtual machines, MySQL database, as well as a NFS capable of serving 1,000 people visiting the blog per day will cost you around $70 USD per month on AWS. That is all you need to come to a decision.

Because the infrastructure does not contain any important data and you have finished your evaluation, you can delete all the resources and stop paying for them.

Go to the CloudFormation service in the Management Console, and take the following steps in figure 2.19:

1. Select the check box at the beginning of the row containing your wordpress stack.
2. Open the Actions menu by clicking Actions.
3. Click Delete Stack.

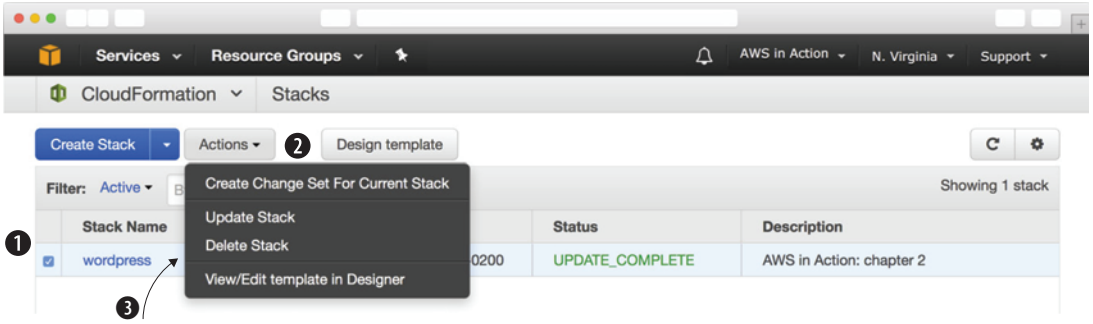


Figure 2.19 Delete your blogging infrastructure.

After you confirm the deletion of the infrastructure, as shown in figure 2.20, it takes a few minutes for AWS to delete all of the infrastructure's dependencies.

This is an efficient way to manage your infrastructure. Just as the infrastructure's creation was automated, its deletion is also. You can create and delete infrastructure on-demand whenever you like. You only pay for infrastructure when you create and run it.

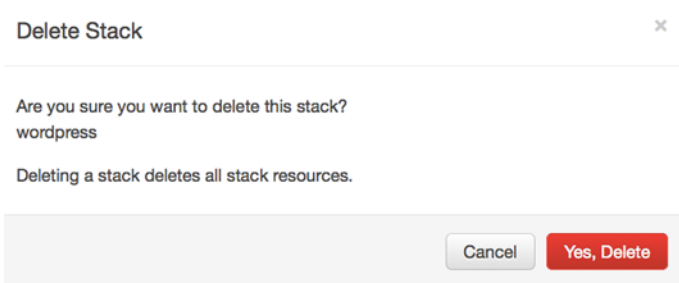


Figure 2.20 Confirming deletion of your blogging infrastructure

Summary

- Creating a blogging infrastructure can be fully automated.
- Infrastructure can be created at any time on-demand, without any up-front commitment for how long you'll use it.
- You pay for your infrastructure based on usage. For example, you are paying for a virtual machine per second of usage.
- Infrastructure consists of several parts, such as virtual machines, load balancers, and databases.
- Infrastructure can be deleted with one click. The process is powered by automation.

Part 2

Building virtual infrastructure consisting of computers and networking

Computing power and network connectivity has become a basic need for private households, medium-sized enterprises, and big corporations. Operating hardware in data centers that are in-house or outsourced has covered these needs in the past. Now the cloud is revolutionizing the way you can access computing power.

Virtual machines can be started and stopped on-demand to fulfill your computing needs within minutes. Being able to install software on virtual machines enables you to execute your computing tasks without needing to buy or rent hardware.

If you want to understand AWS, you have to dive into the possibilities of the API working behind the scenes. You can control every single service on AWS by sending requests to a REST API. Based on this, there is a variety of solutions that help you to automate your overall infrastructure. Infrastructure automation is a big advantage of the cloud compared to hosting on-premises. This chapter will guide you into infrastructure orchestration and the automated deployment of applications.

Creating *virtual networks* allows you to build closed and secure network environments on AWS and to connect these networks with your home or corporate network.

Chapter 3 covers working with virtual machines. You will learn about the key concepts of the EC2 service.

Chapter 4 contains different approaches to automate your infrastructure. You will learn how to make use of Infrastructure-as-Code.

Chapter 5 shows three ways of deploying your software to AWS.

Chapter 6 is about networking. You will learn how to secure your system with a virtual private network and firewalls.

Chapter 7 is about a new way of computing: functions. You will learn how to automate operational tasks with AWS Lambda.

3 Using virtual machines: EC2

This chapter covers

- Launching a virtual machine with Linux
- Controlling a virtual machine remotely via SSH
- Monitoring and debugging a virtual machine
- Saving costs for virtual machines

It's impressive what you can achieve with the computing power of the smartphone in your pocket or the laptop in your bag. But if your task requires massive computing power or high network traffic, or needs to run reliably 24/7, a virtual machine is a better fit. With a virtual machine, you get access to a slice of a physical machine located in a data center. On AWS, virtual machines are offered by the service called Elastic Compute Cloud (EC2).

Not all examples are covered by Free Tier

The examples in this chapter are *not* all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

3.1 Exploring a virtual machine

A virtual machine (VM) is a part of a physical machine that's isolated by software from other VMs on the same physical machine; it consists of CPUs, memory, networking interfaces, and storage. The physical machine is called the *host machine*, and the VMs running on it are called *guests*. A *hypervisor* is responsible for isolating the guests from each other and for scheduling requests to the hardware, by providing a virtual hardware platform to the guest system. Figure 3.1 shows these layers of virtualization.

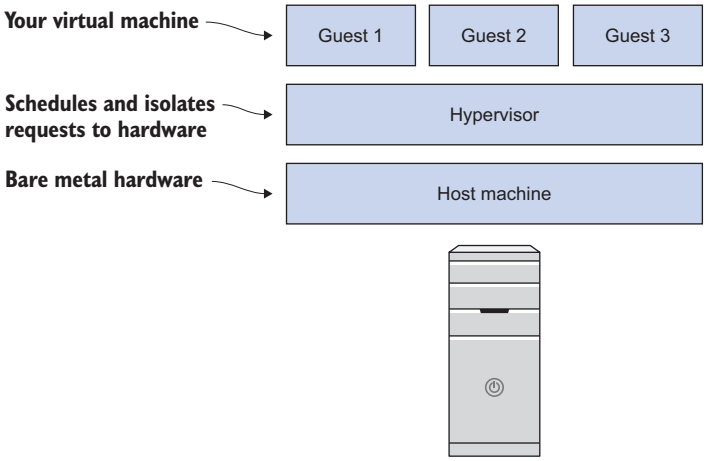


Figure 3.1 Layers of virtualization

Typical use cases for a virtual machine are as follows:

- Hosting a web application such as WordPress
- Operating an enterprise application, such as an ERP application
- Transforming or analyzing data, such as encoding video files

3.1.1 Launching a virtual machine

It takes only a few clicks to launch a virtual machine:

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Make sure you're in the N. Virgina (US East) region (see figure 3.2), because we optimized our examples for this region.
3. Find the EC2 service in the navigation bar under Services, and click it. You'll see a page like the one in figure 3.3.
4. Click Launch Instance to start the wizard for launching a virtual machine.

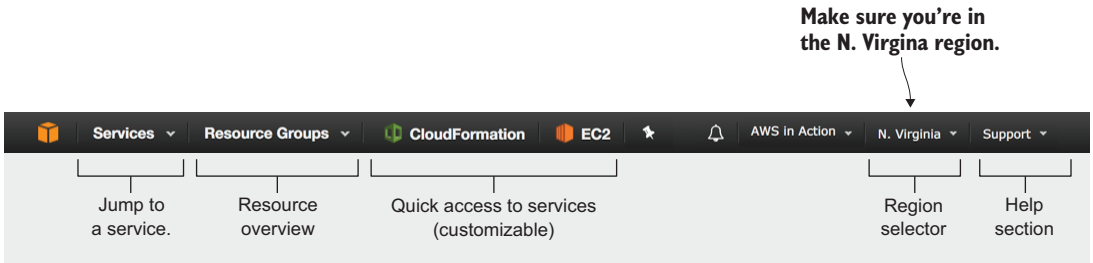


Figure 3.2 Examples are optimized for the N. Virginia region

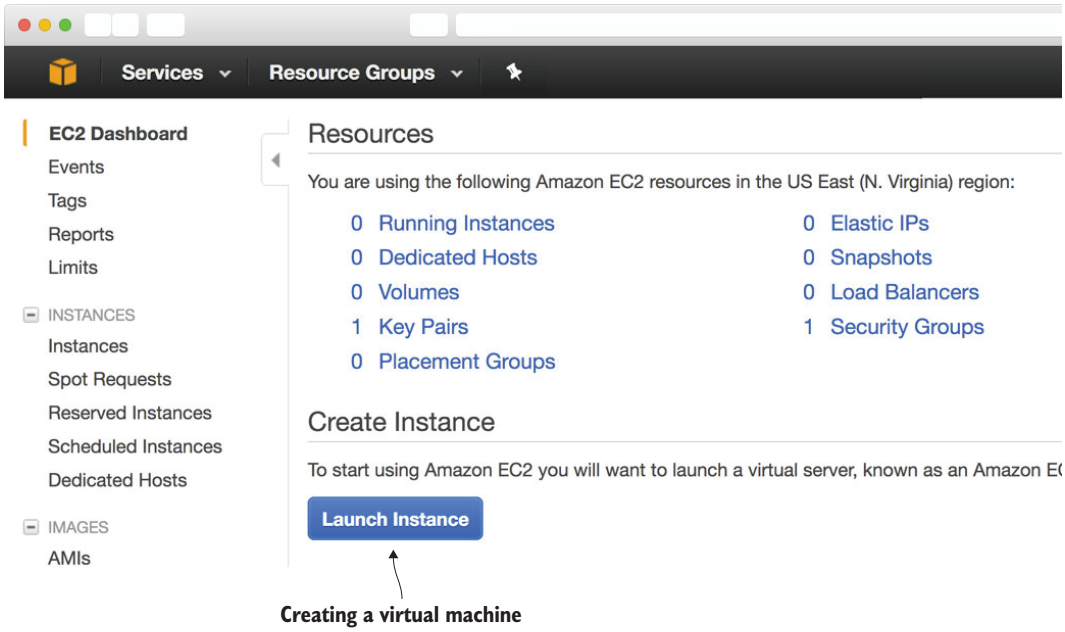


Figure 3.3 The EC2 Dashboard gives you an overview of all parts of the service.

The wizard in figure 3.3 will guide you through the following steps:

1. Selecting an OS
2. Choosing the size of your virtual machine
3. Configuring details
4. Adding storage
5. Tagging your virtual machine
6. Configuring a firewall
7. Reviewing your input and selecting a key pair used for SSH authentication

SELECTING THE OPERATING SYSTEM

The first step is to choose an OS. In AWS, the OS comes bundled with preinstalled software for your virtual machine; this bundle is called an *Amazon Machine Image (AMI)*. Select Ubuntu Server 16.04 LTS (HVM), as shown in figure 3.4. Why Ubuntu? Because Ubuntu offers a ready-to-install package named *linkchecker* that you'll use later to check a website for broken links.

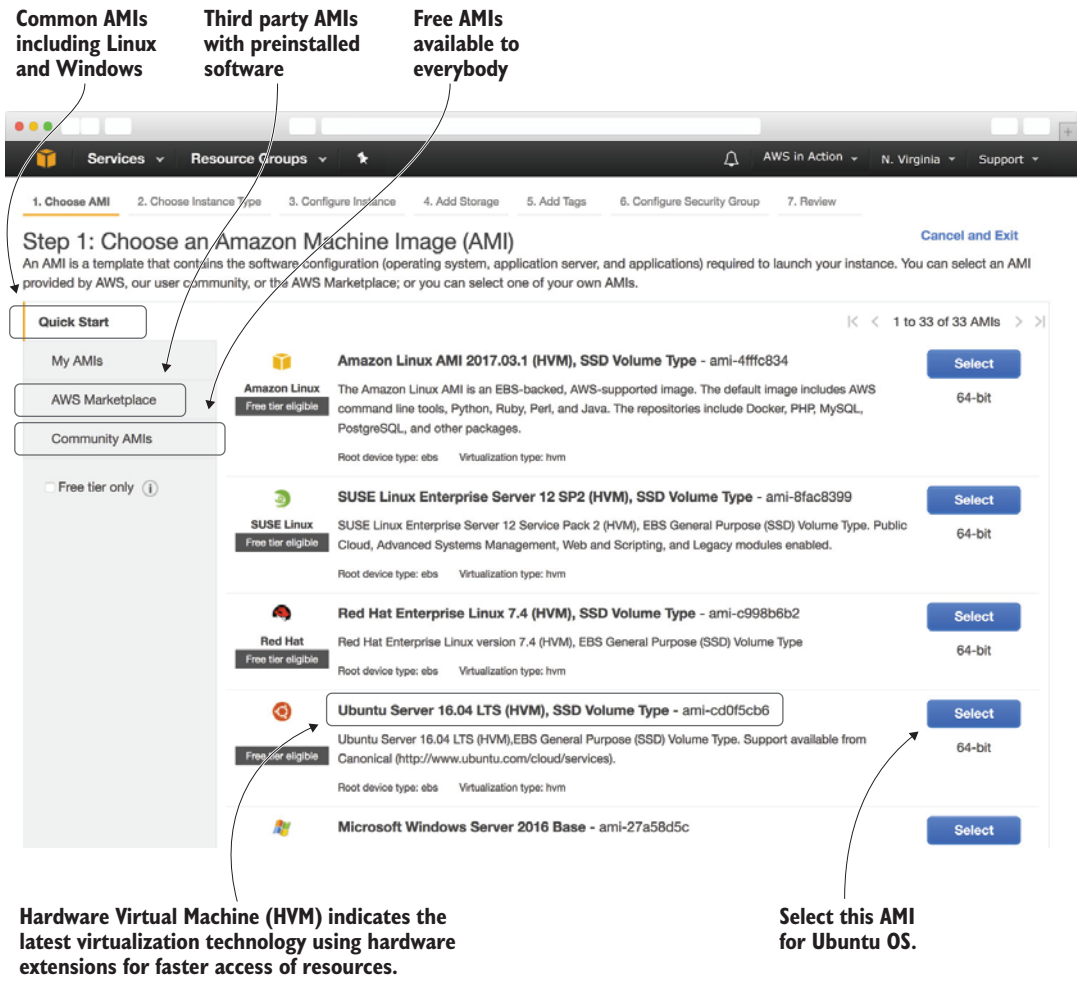


Figure 3.4 Choose the OS for your virtual machine.

The AMI is the basis your virtual machine starts from. AMIs are offered by AWS, third-party providers, and by the community. AWS offers the Amazon Linux AMI, which is based on Red Hat Enterprise Linux and optimized for use with EC2. You'll also find popular Linux distributions and AMIs with Microsoft Windows Server, and you can find more AMIs with preinstalled third-party software in the AWS Marketplace.

Amazon Linux 2 is coming

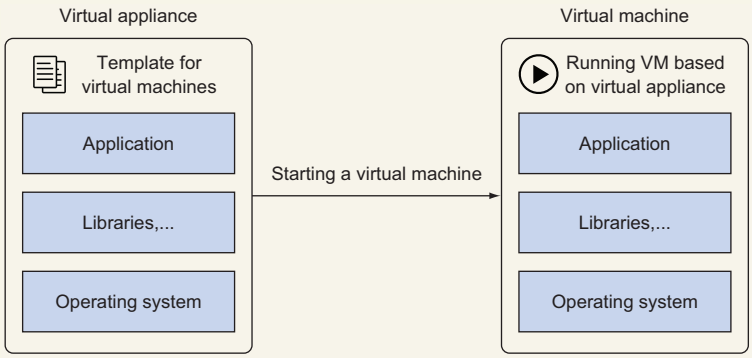
Amazon Linux 2 is the next generation Amazon Linux OS. At the time of writing, no LTS release of Amazon Linux 2 was available. But now, as you read this book, that might have changed. Check out the Amazon Linux 2 FAQs: https://aws.amazon.com/amazon-linux-2/faqs/.

Amazon Linux 2 adds long-term support for five years. You can now run Amazon Linux 2 locally and on-premises as well. Besides that, systemd is now used and a new mechanism called the *extras library* provides up-to-date versions of software bundles such as NGINX.

When choosing an AMI, start by thinking about the requirements of the application you want to run on the VM. Your knowledge and experience with a specific operating system is another important factor when deciding which AMI to start with. It's also important that you trust the AMI's publisher. We prefer working with Amazon Linux, as it is maintained and optimized by AWS.

Virtual appliances on AWS

A *virtual appliance* is an image of a virtual machine containing an OS and preconfigured software. Virtual appliances are used when the hypervisor starts a new VM. Because a virtual appliance contains a fixed state, every time you start a VM based on a virtual appliance, you'll get exactly the same result. You can reproduce virtual appliances as often as needed, so you can use them to eliminate the cost of installing and configuring complex stacks of software. Virtual appliances are used by virtualization tools from VMware, Microsoft, and Oracle, and for infrastructure-as-a-service (IaaS) offerings in the cloud.



A virtual appliance contains a template for a virtual machine

The AMI is a special type of virtual appliance for use with the EC2 service. An AMI technically consists of a read-only filesystem including the OS, additional software, and configuration; it doesn't include the kernel of the OS. The kernel is loaded from an Amazon Kernel Image (AKI). You can also use AMIs for deploying software on AWS.

(continued)

AWS uses Xen, an open source hypervisor. The current generations of VMs on AWS use hardware-assisted virtualization. The technology is called Hardware Virtual Machine (HVM). A virtual machine run by an AMI based on HVM uses a fully virtualized set of hardware, and can take advantage of extensions that provide fast access to the underlying hardware.

Using a version 3.8+ kernel for your Linux-based VMs will provide the best performance. To do so, you should use at least Amazon Linux 13.09, Ubuntu 14.04, or RHEL7. If you're starting new VMs, make sure you're using HVM images.

In November 2017 AWS announced a new generation of virtualization called Nitro. Nitro combines a KVM-based hypervisor with customized hardware (ASICs) aiming to provide a performance that is indistinguishable from bare metal machines. Currently, the c5 and m5 instance types make use of Nitro, and it's likely that more instance families using Nitro will follow.

CHOOSING THE SIZE OF YOUR VIRTUAL MACHINE

It's now time to choose the computing power needed for your virtual machine. Figure 3.5 shows the next step of the wizard. On AWS, computing power is classified into instance types. An instance type primarily describes the number of virtual CPUs and the amount of memory.

Table 3.1 shows examples of instance types for different use cases. The prices represent the actual prices in USD for a Linux VM in the US East (N. Virginia) region, as recorded Aug. 31, 2017.

Table 3.1 Examples of instance families and instance types

Instance type	Virtual CPUs	Memory	Description	Typical use case	Hourly cost (USD)
t2.nano	1	0.5 GB	Smallest and cheapest instance type, with moderate baseline performance and the ability to burst CPU performance above the baseline	Testing and development environments, and applications with very low traffic	0.0059
m4.large	2	8 GB	Has a balanced ratio of CPU, memory, and networking performance	All kinds of applications, such as medium databases, web servers, and enterprise applications	0.1
r4.large	2	15.25 GB	Optimized for memory-intensive applications with extra memory	In-memory caches and enterprise application servers	0.133

There are instance families optimized for different kinds of use cases.

- *T family*—Cheap, moderate baseline performance with the ability to burst to higher performance for short periods of time
- *M family*—General purpose, with a balanced ration of CPU and memory

- *C family*—Computing optimized, high CPU performance
- *R family*—Memory optimized, with more memory than CPU power compared to M family
- *D family*—Storage optimized, offering huge HDD capacity
- *I family*—Storage optimized, offering huge SSD capacity
- *X family*—Extensive capacity with a focus on memory, up to 1952 GB memory and 128 virtual cores
- *F family*—Accelerated computing based on FPGAs (field programmable gate arrays)
- *P, G, and CG family*—Accelerated computing based on GPUs (graphics processing units)

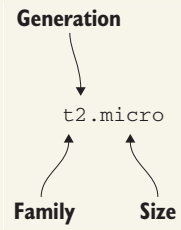
Our experience indicates that you'll overestimate the resource requirements for your applications. So, we recommend that you try to start your application with a smaller instance type than you think you need at first. You can change the instance family and type later if needed.

Instance types and families

The names for different instance types are all structured in the same way. The *instance family* groups instance types with similar characteristics. AWS releases new instance types and families from time to time; the different versions are called *generations*. The *instance size* defines the capacity of CPU, memory, storage, and networking.

The instance type *t2.micro* tells you the following:

- The instance family is called *t*. It groups small, cheap virtual machines with low baseline CPU performance but the ability to burst significantly over baseline CPU performance for a short time.
- You're using generation *2* of this instance family.
- The size is *micro*, indicating that the EC2 instance is very small.



Disassembling the t2.micro instance type

Computer hardware is getting faster and more specialized, so AWS is constantly introducing new instance types and families. Some of them are improvements of existing instance families, and others are focused on specific workloads. For example, the instance family R4 was introduced in November 2016. It provides instances for memory-intensive workloads and improves the R3 instance types.

One of the smallest and cheapest VMs will be enough for your first experiments. In the wizard screen shown in figure 3.5, choose the instance type t2.micro, which is eligible for the Free Tier. Then click Next: Configure Instance Details to proceed.

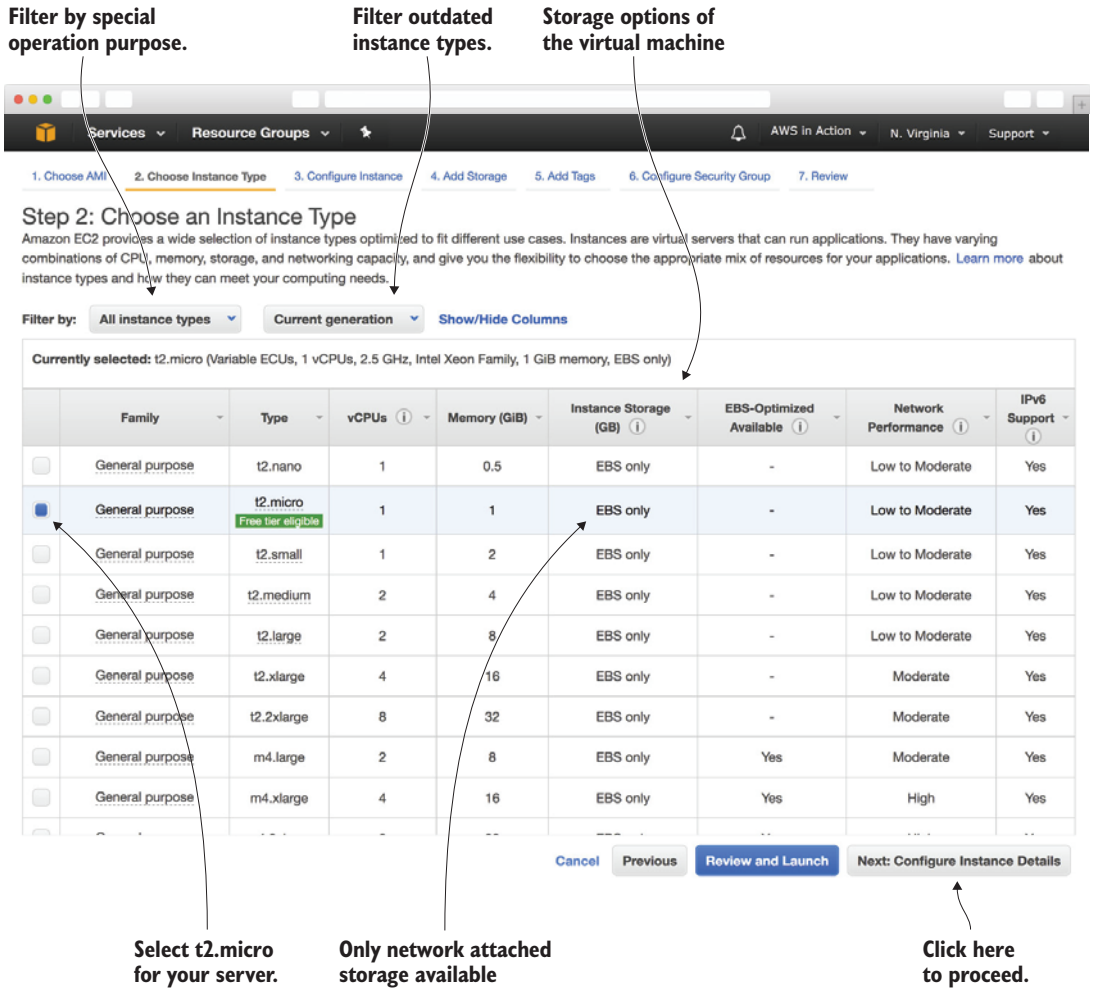


Figure 3.5 Choosing the size of your virtual machine

CONFIGURING DETAILS, STORAGE, FIREWALL, AND TAGS

The next four steps of the wizard (shown in figure 3.6–figure 3.9) are easy, because you don't need to change the defaults. You'll learn about these settings in detail later in the book.

Figure 3.6 shows where you can change the details for your VM, such as the network configuration or the number of VMs to launch. For now, keep the defaults, and click Next: Add Storage to proceed.

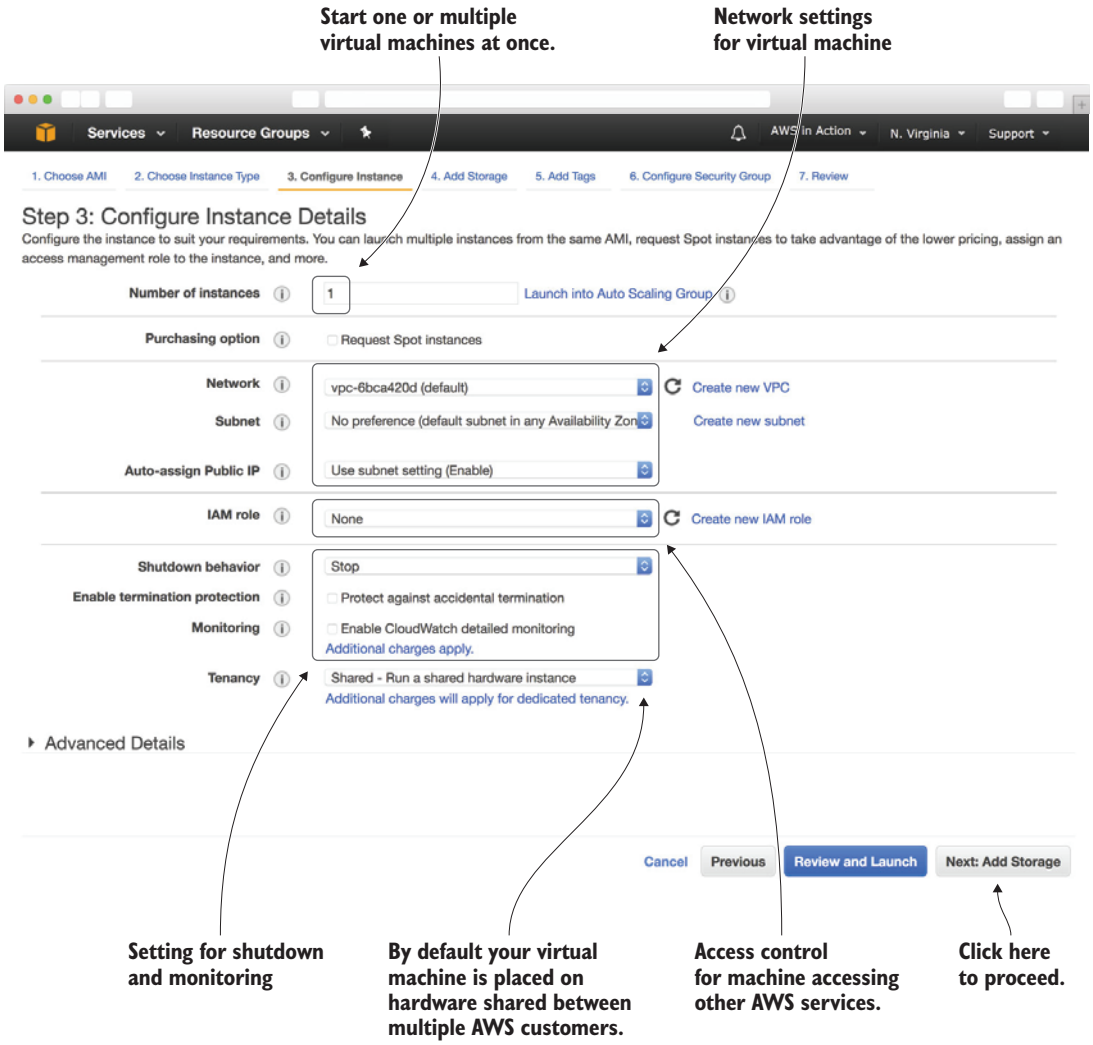


Figure 3.6 Details for the virtual machine

There are different options for storing data on AWS, which we'll cover in detail in the following chapters. Figure 3.7 shows the screen where you can add network-attached storage to your virtual machine. Keep the defaults, and click Next: Add Tags.

A tidy house indicates a tidy mind. Tags help you organize resources on AWS. Figure 3.8 shows the Add Tags screen. Each tag is nothing more than a key-value pair. For this example, add at least a `Name` tag to help you find your stuff later. Use `Name` as the key and `mymachine` as the value, as shown in the figure. Then click Next: Configure Security Group.

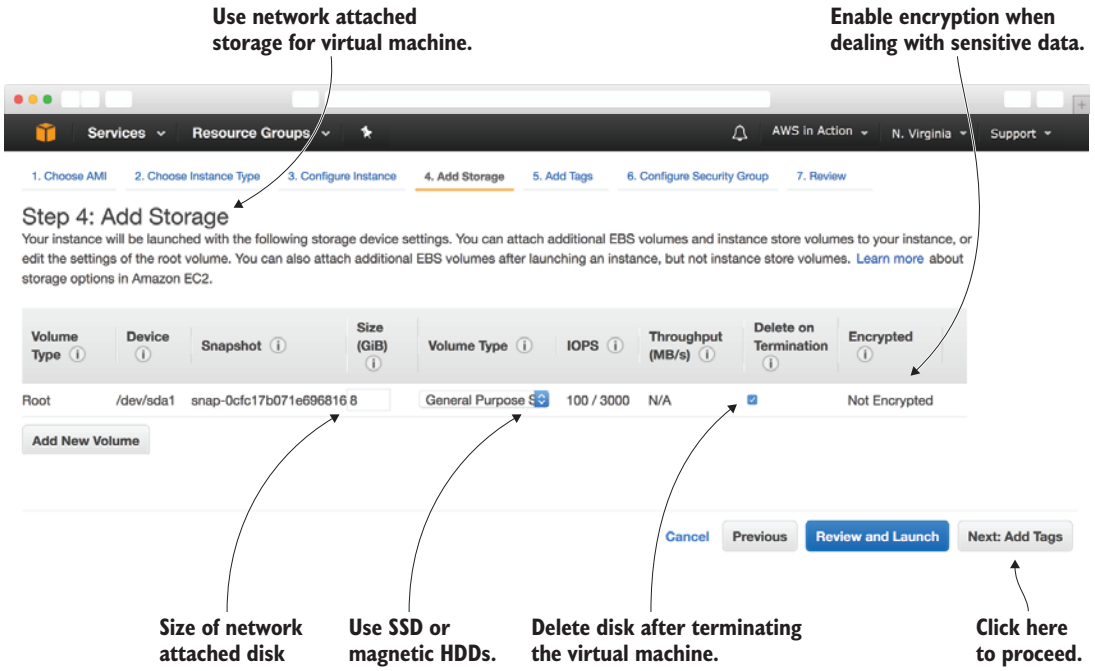


Figure 3.7 Adding network-attached storage to your virtual machine

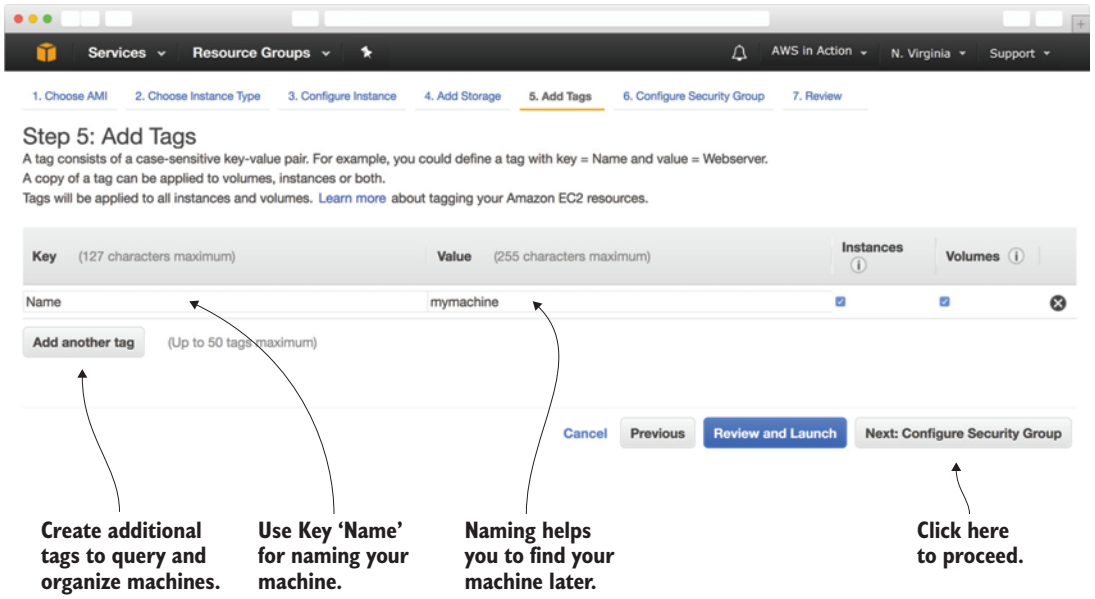


Figure 3.8 Tagging your virtual machine with a `Name` tag

Organizing AWS resources with tags

Most AWS resources can be tagged. For example, you can add tags to an EC2 instance. There are three major use cases for resource tagging:

1. Use tags to filter and search for resources.
2. Analyze your AWS bill based on resource tags.
3. Restrict access to resources based on tags.

Typical tags include environment type (such as test or production), responsible team, department, and cost center.

A firewall helps to secure your virtual machine. Figure 3.9 shows the settings for a default firewall allowing access via SSH from anywhere.

1. Select Create a new security group.
2. Type in `ssh-only` for the name and description of the security group.
3. Keep the default rule allowing SSH from anywhere.
4. Click Review and Launch to proceed with the next step.

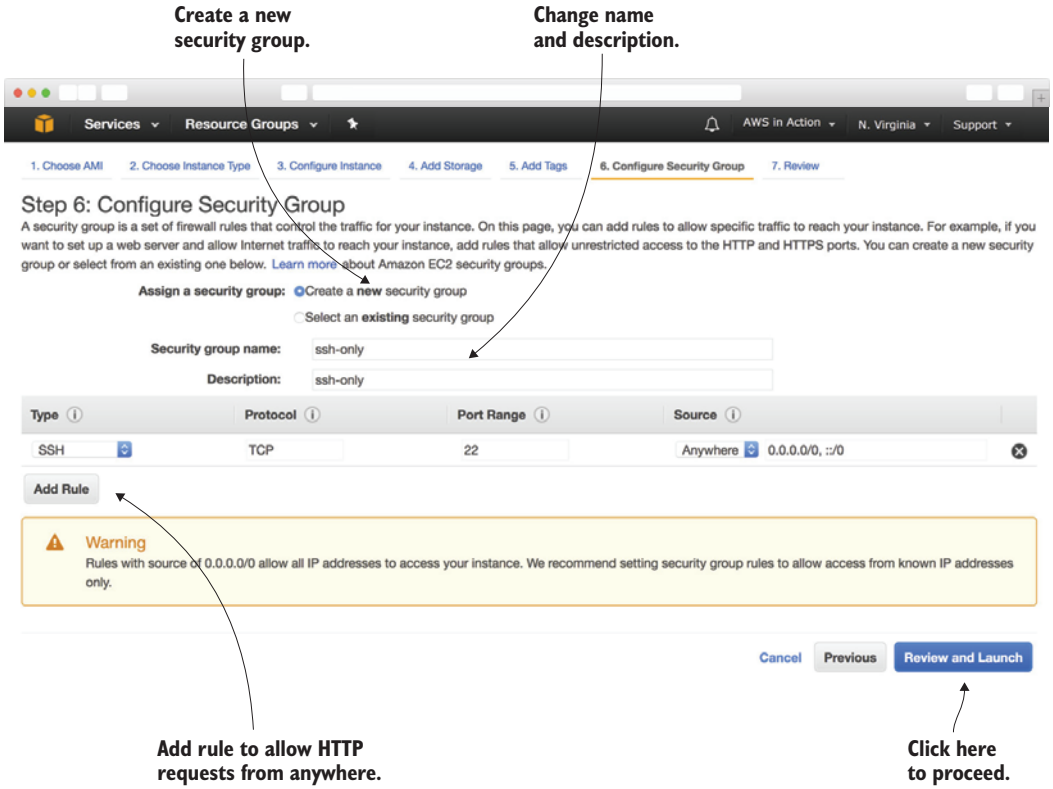


Figure 3.9 Configuring the firewall for your virtual machine

REVIEWING YOUR INPUT AND SELECTING A KEY PAIR FOR SSH

You're almost finished. The wizard should show a review of your new virtual machine (see figure 3.10). Make sure you chose Ubuntu Server 16.04 LTS (HVM) as the OS and t2.micro as the instance type. If everything is fine, click the Launch button. If not, go back and make changes to your VM where needed.

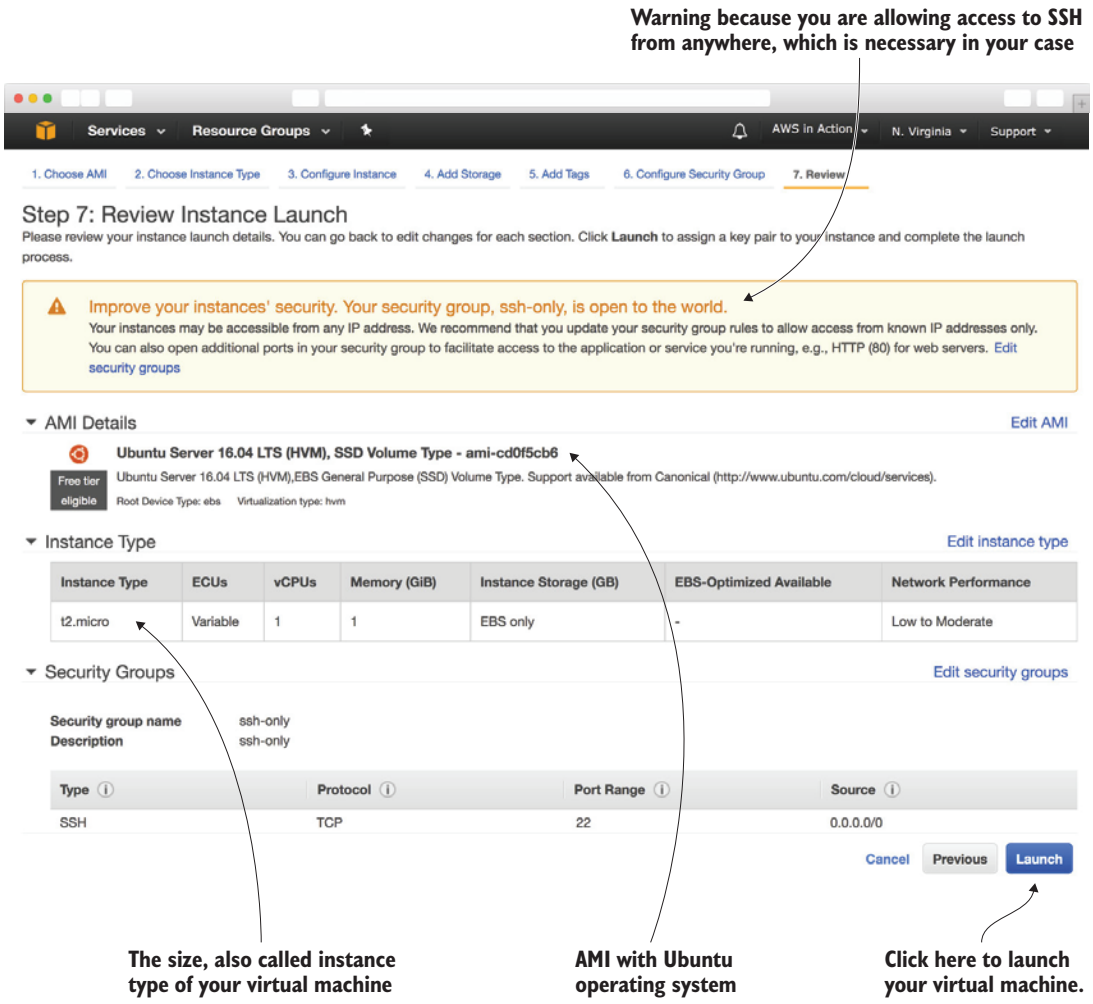


Figure 3.10 Review launch for the virtual machine

Last but not least, the wizard asks for your new virtual machine's key. Choose the option Choose an Existing Key Pair, select the key pair `mykey`, and click Launch Instances (see figure 3.11).

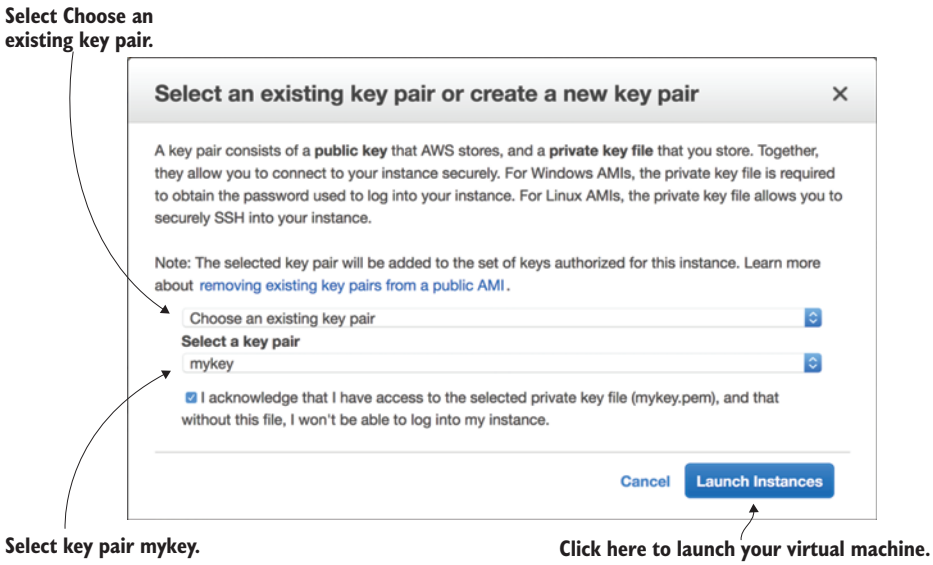


Figure 3.11 Choosing a key pair for your virtual machine

Missing your key?

Logging in to your virtual machine requires a key. You use a key instead of a password to authenticate yourself. Keys are much more secure than passwords, and using keys for SSH is enforced for VMs running Linux on AWS. If you skipped the creation of a key in section 1.8.3, follow these steps to create a personal key:

1. Open the AWS Management Console at https://console.aws.amazon.com. Find the EC2 service in the navigation bar under Services, and click it.
2. Switch to Key Pairs via the submenu.
3. Click Create Key Pair.
4. Enter `mykey` for Key Pair Name, and click Create. Your browser downloads the key automatically.
5. Open your terminal, and switch to your download folder.

6a. Linux and macOS only: change the access rights of the file mykey.pem by running `chmod 400 mykey.pem` in your terminal.

6b. Windows only: Windows doesn't ship an SSH client yet, so you need to install PuTTY. PuTTY comes with a tool called PuTTYgen that can convert the mykey.pem file into a mykey.ppk file, which you'll need. Open PuTTYgen, and select SSH-2 RSA under Type of Key to Generate. Click Load. Because PuTTYgen displays only *.ppk files, you need to switch the file extension of the File Name Input to All Files. Now you can select the mykey.pem file and click Open. Click OK in the confirmation dialog box. Change Key Comment to `mykey`. Click Save Private Key. Ignore the warning about saving the key without a passphrase. Your .pem file is now converted to the .ppk format needed by PuTTY.

You'll find a more detailed explanation about how to create a key in chapter 1.

Your virtual machine should now launch. Open an overview by clicking View Instances, and wait until the machine reaches the Running state. To take full control over your virtual machine, you need to log in remotely.

3.1.2 Connecting to your virtual machine

Installing additional software and running commands on your virtual machine can be done remotely. To log in to the virtual machine, you have to figure out its public domain name:

1. Click the EC2 service in the navigation bar under Services, and click Instances in the submenu at left to jump to an overview of your virtual machine.

   Select the virtual machine from the table by clicking it. Figure 3.12 shows the overview of your virtual machines and the available actions.

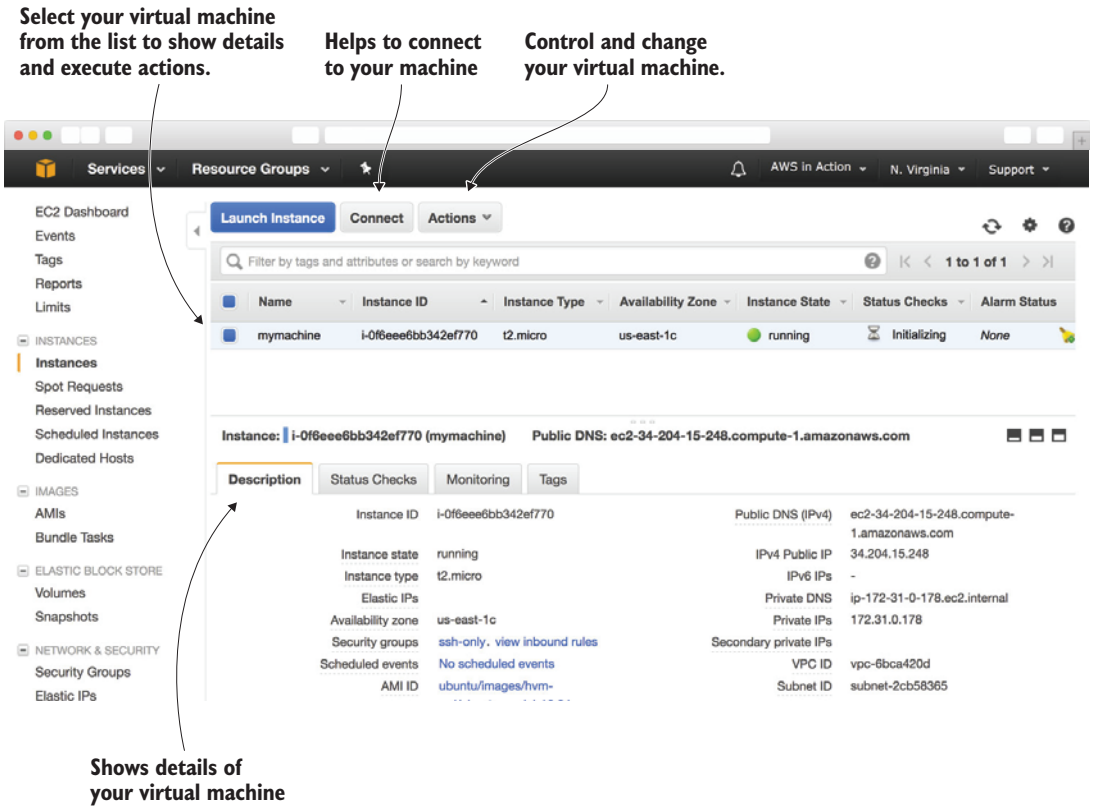


Figure 3.12 Overview of your virtual machines with actions to control them

2. Click Connect to open the instructions for connecting to the virtual machine.

   Figure 3.13 shows the dialog with instructions for connecting to the virtual machine. Find the public DNS of your virtual machine, such as `ec2-34-204-15-248.compute-1.amazonaws.com` in our example.

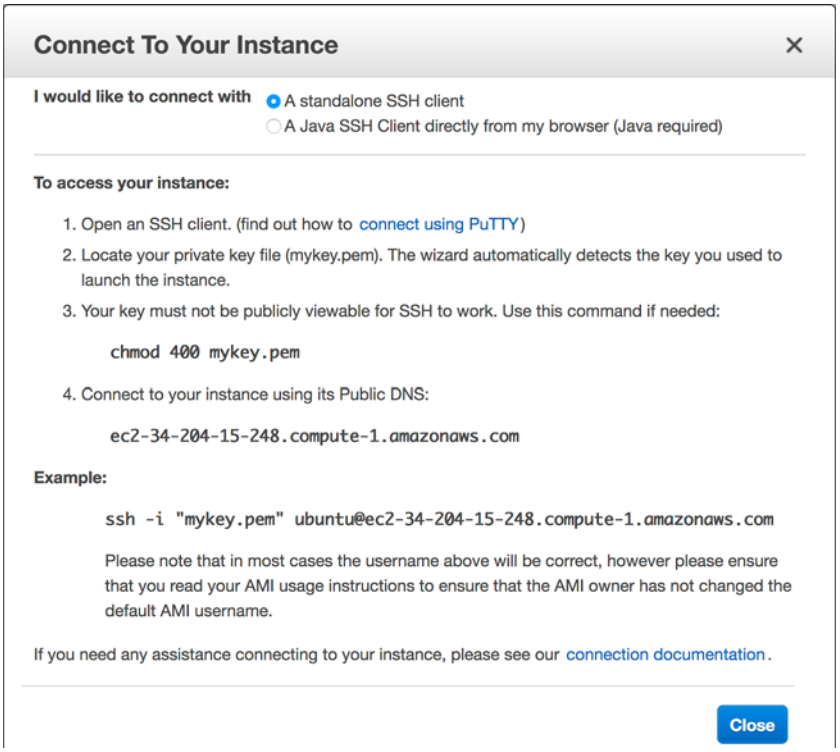


Figure 3.13 Instructions for connecting to the virtual machine with SSH

With the public DNS and your key, you can connect to your virtual machine. Continue to the next section, depending on your OS.

Linux and macOS

Open your terminal, and type `ssh -i $PathToKey/mykey.pem ubuntu@$PublicDns`, replacing *`$PathToKey`* with the path to the key file you downloaded in section 1.8.3 and *`$PublicDns`* with the public DNS shown in the Connect dialog in the AWS Management Console. You'll see a security alert regarding the authenticity of the new host. Answer yes to connect.

Windows

For Windows, follow these steps:

1. Find the mykey.ppk file you created in section 1.8.3, and double-click to open it.
2. PuTTY Pageant should appear in the Windows taskbar as an icon. If not, you may need to install or reinstall PuTTY as described in section 1.8.3.
3. Start PuTTY. Fill in the public DNS shown in the Connect dialog in the AWS Management Console, and click Open (see figure 3.14).
4. You'll see a security alert regarding the authenticity of the new host. Answer yes and type `ubuntu` as the login name. Click Enter.

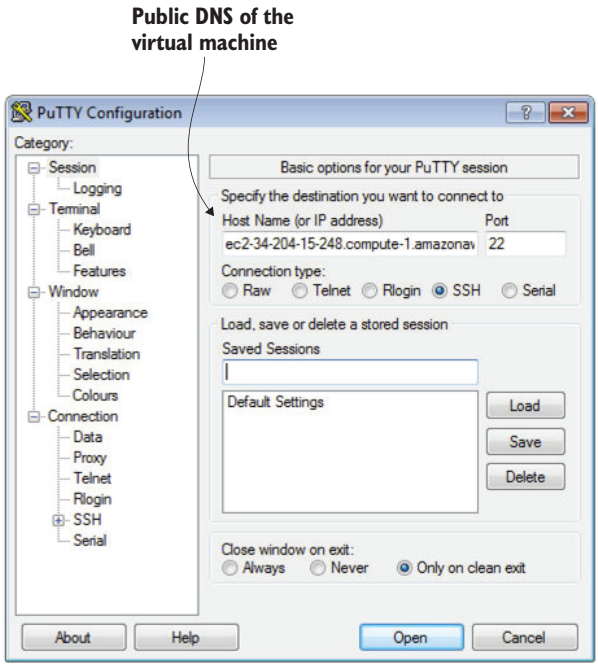


Figure 3.14 Connecting to the virtual machine with PuTTY on Windows

LOGIN MESSAGE

Whether you're using Linux, Mac OS, or Windows, after a successful login you should see a message like the following:

```
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1022-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-0-178:~$
```

You're now connected to your virtual machine and are ready to run a few commands.

3.1.3 Installing and running software manually

You've now started a virtual machine with an Ubuntu OS. It's easy to install additional software with the help of the package manager apt. First we need to make sure the package manager is up-to-date. Please run the following command to update the list of available packages:

```
$ sudo apt-get update
```

To begin, you'll install a tiny tool called `linkchecker` that allows you to find broken links on a website:

```
$ sudo apt-get install linkchecker -y
```

Now you're ready to check for links pointing to websites that no longer exist. To do so, choose a website and run the following command:

```
$ linkchecker https://...
```

The output of checking the links looks something like this:

```
[...]
URL        `http://www.linux-mag.com/blogs/fableson'
Name       `Frank Ableson's Blog'
Parent URL http://manning.com/about/blogs.html, line 92, col 27
Real URL   http://www.linux-mag.com/blogs/fableson
Check time 1.327 seconds
Modified   2015-07-22 09:49:39.000000Z
Result     Error: 404 Not Found

URL        `/catalog/dotnet'
Name       `Microsoft & .NET'
Parent URL http://manning.com/wittig/, line 29, col 2
Real URL   http://manning.com/catalog/dotnet/
Check time 0.163 seconds
D/L time   0.146 seconds
Size       37.55KB
Info       Redirected to `http://manning.com/catalog/dotnet/'.
           235 URLs parsed.
Modified   2015-07-22 01:16:35.000000Z
Warning    [http-moved-permanent] HTTP 301 (moved permanent)
           encountered: you should update this link.
Result     Valid: 200 OK
[...]
```

Depending on the number of web pages, the crawler may need some time to check all of them for broken links. At the end, it lists the broken links and gives you the chance to find and fix them.

3.2 Monitoring and debugging a virtual machine

If you need to find the reason for an error or why your application isn't behaving as you expect, it's important to have access to tools that can help with monitoring and debugging. AWS provides tools that let you monitor and debug your virtual machines. One approach is to examine the virtual machine's logs.

3.2.1 Showing logs from a virtual machine

If you need to find out what your virtual machine was doing during and after startup, there is a simple solution. AWS allows you to see the EC2 instance's logs with the help of the Management Console (the web interface you use to start and stop virtual machines). Follow these steps to open your VM's logs:

1. Open the EC2 service from the main navigation, and select Instances from the submenu.
2. Select the running virtual machine by clicking the row in the table.
3. In the Actions menu, choose Instance Settings > Get System Log.

A window opens and shows you the system logs from your VM that would normally be displayed on a physical monitor during startup (see figure 3.15).

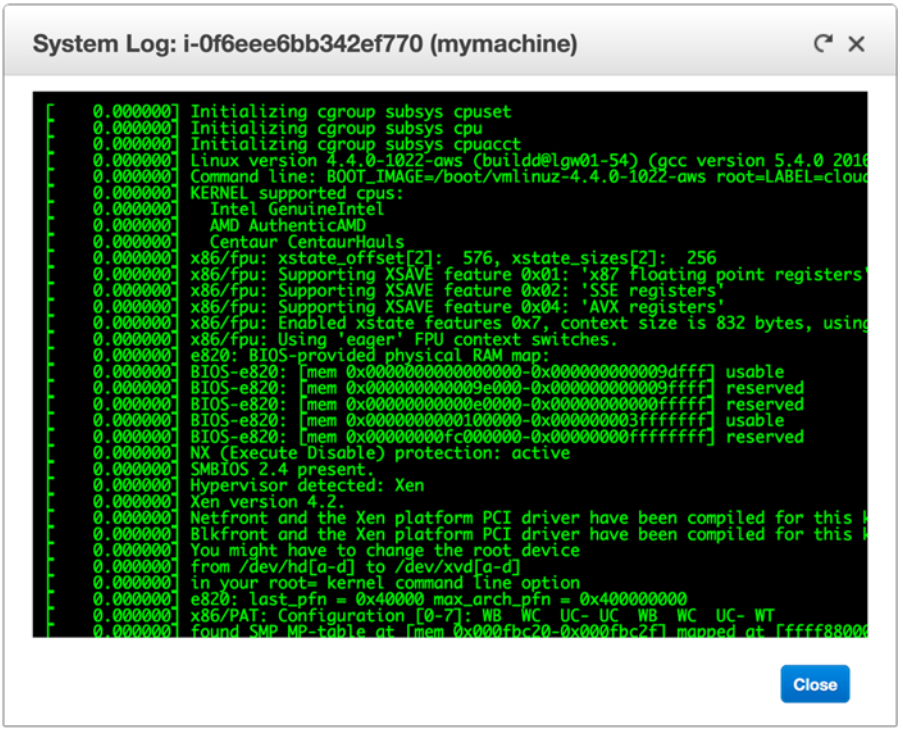


Figure 3.15 Debugging a virtual machine with the help of logs

The log contains all log messages that would be displayed on the monitor of your machine if you were running it on-premises. Watch out for any log messages stating that an error occurred during startup. If the error message is not obvious, you should contact the vendor of the AMI, AWS Support, or post your question in the AWS Developer Forums at https://forums.aws.amazon.com.

This is a simple and efficient way to access your system logs without needing an SSH connection. Note that it will take several minutes for a log message to appear in the log viewer.

3.2.2 Monitoring the load of a virtual machine

AWS can help you answer another question: is your virtual machine close to its maximum capacity? Follow these steps to open the EC2 instance's metrics:

1. Open the EC2 service from the main navigation, and select Instances from the submenu.
2. Select the running virtual machine by clicking the appropriate row in the table.
3. Select the Monitoring tab at lower right.
4. Click the Network In chart to dive into the details.

You'll see a graph that shows the virtual machine's utilization of incoming networking traffic, similar to figure 3.16. There are metrics for CPU, network, and disk usage. As AWS is looking at your VM from the outside, there is no metric indicating the memory usage. You can publish a memory metric yourself, if needed. The metrics are updated every 5 minutes if you use basic monitoring, or every minute if you enable detailed monitoring of your virtual machine, which costs extra.

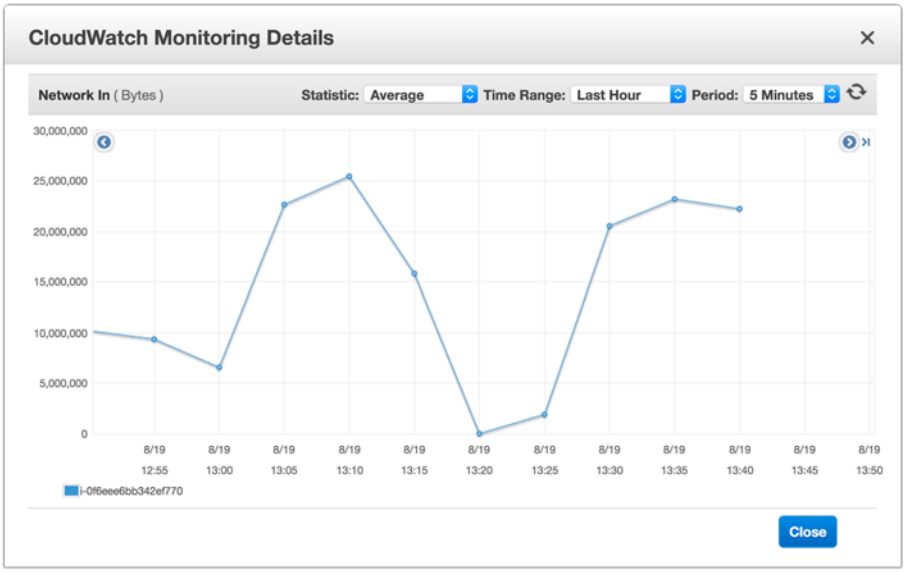


Figure 3.16 Gaining insight into a virtual machine's incoming network traffic with the CloudWatch metric

Checking the metrics of your EC2 instance is helpful when debugging performance issues. You will also learn how to increase or decrease your infrastructure based on these metrics in chapter 17.

Metrics and logs help you monitor and debug your virtual machines. Both tools can help ensure that you're providing high-quality services in a cost-efficient manner. Look at Monitoring Amazon EC2 in the AWS documentation at http://mng.bz/0q40 if you are looking for more detailed information about monitoring your virtual machines.

3.3 Shutting down a virtual machine

To avoid incurring charges, you should always turn off virtual machines you're not using them. You can use the following four actions to control a virtual machine's state:

- *Start*—You can always start a stopped virtual machine. If you want to create a completely new machine, you'll need to launch a virtual machine.
- *Stop*—You can always stop a running virtual machine. A stopped virtual machine doesn't incur charges, expect for attached resources like network-attached storage. A stopped virtual machine can be started again but likely on a different host. If you're using network-attached storage, your data persists.
- *Reboot*—Have you tried turning it off and on again? If you need to reboot your virtual machine, this action is what you want. You won't lose any persistent data when rebooting a virtual machine because it stays on the same host.
- *Terminate*—Terminating a virtual machine means deleting it. You can't start a virtual machine that you've already terminated. The virtual machine is deleted, usually together with dependencies like network-attached storage and public and private IP addresses. A terminated virtual machine doesn't incur charges.

WARNING The difference between *stopping* and *terminating* a virtual machine is important. You can start a stopped virtual machine. This isn't possible with a terminated virtual machine. If you terminate a virtual machine, you delete it.

Figure 3.17 illustrates the difference between stopping and terminating an EC2 instance, with the help of a flowchart.

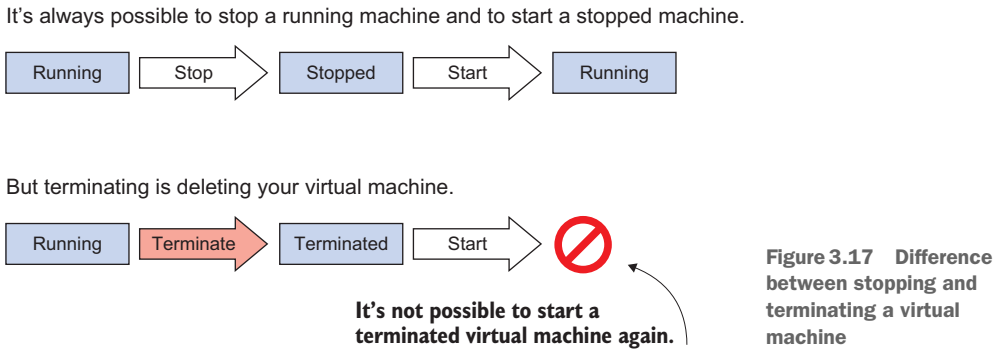


Figure 3.17 Difference between stopping and terminating a virtual machine

Stopping or terminating unused virtual machines saves costs and prevents you from being surprised by an unexpected bill from AWS. You may want to stop or terminate unused virtual machines when:

- *You have launched virtual machines to implement a proof-of-concept.* After finishing the project, the virtual machines are no longer needed. Therefore, you can terminate them.
- *You are using a virtual machine to test a web application.* As no one else uses the virtual machine, you can stop it before you knock off work, and start it back up again the following day.
- *One of your customers canceled their contract.* After backing up relevant data, you can terminate the virtual machines that had been used for your former customer.

After you terminate a virtual machine, it's no longer available and eventually disappears from the list of virtual machines.

Cleaning up

Terminate the virtual machine named `mymachine` that you started at the beginning of this chapter:

1. Open the EC2 service from the main navigation, and select Instances from the submenu.
2. Select the running virtual machine by clicking the row in the table.
3. In the Actions menu, choose Instance State > Terminate.

3.4 Changing the size of a virtual machine

It is always possible to change the size of a virtual machine. This is one of the advantages of using the cloud, and it gives you the ability to scale vertically. If you need more computing power, increase the size of the EC2 instance.

In this section, you'll learn how to change the size of a running virtual machine. To begin, follow these steps to start a small virtual machine:

1. Open the AWS Management Console, and choose the EC2 service.
2. Start the wizard to launch a new virtual machine by clicking Launch Instance.
3. Select Ubuntu Server 16.04 LTS (HVM) as the AMI for your virtual machine.
4. Choose the instance type t2.micro.
5. Click Review and Launch to start the virtual machine.
6. Click Edit Security Groups to configure the firewall. Choose Select an Existing Security Group and select the security group named `ssh-only`.
7. Click Review and Launch to start the virtual machine.
8. Check the summary for the new virtual machine, and click Launch.

9. Choose the option Choose an Existing Key Pair, select the key pair `mykey`, and click Launch Instances.
10. Switch to the overview of EC2 instances, and wait for the new virtual machine's state to switch to Running.

You've now started an EC2 instance of type t2.micro. This is one of the smallest virtual machines available on AWS.

Use SSH to connect to your virtual machine, as shown in the previous section, and execute `cat /proc/cpuinfo` and `free -m` to see information about the machine's capabilities. The output should look similar to this:

```
$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 63
model name  : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
stepping    : 2
microcode   : 0x36
cpu MHz     : 2400.054
cache size  : 30720 KB
[...]

$ free -m
              total        used        free      shared  buff/cache   available
Mem:            990          42         632           3         315         794
Swap:             0           0           0
```

Your virtual machine provides a single CPU core and 990 MB of memory. If your application is having performance issues, increasing the instance size can solve the problem. Use your machine's metrics as described in section 3.2 to find out if you are running out of CPU or networking capacity. Would your application benefit from additional memory? If so, increasing the instance size will improve the application's performance as well.

If you need more CPUs, more memory, or more networking capacity, there are many other sizes to choose from. You can even change the virtual machine's instance family and generation. To increase the size of your VM, you first need to stop it:

1. Open the AWS Management Console, and choose the EC2 service.
2. Click Instances in the submenu to jump to an overview of your virtual machines.
3. Select your running VM from the list by clicking it.
4. Choose Stop from the Actions menu.

WARNING Starting a virtual machine with instance type `m4.large` incurs charges. Go to http://aws.amazon.com/ec2/pricing if you want to see the current on-demand hourly price for an m4.large virtual machine.

After waiting for the virtual machine to stop, you can change the instance type:

1. Choose Change Instance Type from the Actions menu under Instance Settings. As shown in figure 3.18, a dialog opens in which you can choose the new instance type for your VM.
2. Select m4.large for Instance Type.
3. Save your changes by clicking Apply.

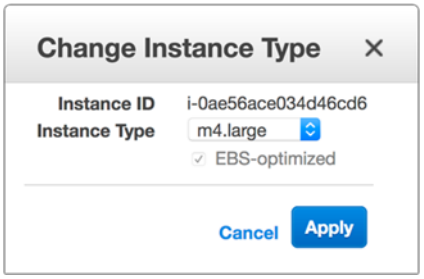


Figure 3.18 Increasing the size of your virtual machine by selecting m4.large as Instance Type

You've now changed the size of your virtual machine and are ready to start it again. To do so, select your virtual machine and choose Start from the Actions menu under Instance State. Your VM will start with more CPUs, more memory, and more networking capabilities. The public and private IP addresses have also changed. Grab the new public DNS to reconnect via SSH; you'll find it in the VM's Details view.

Use SSH to connect to your EC2 instance, and execute `cat /proc/cpuinfo` and `free -m` to see information about its CPU and memory. The output should look similar to this:

```
$ cat /proc/cpuinfo
processor    : 0
vendor_id    : GenuineIntel
cpu family   : 6
model        : 79
model name   : Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
stepping     : 1
microcode    : 0xb00001d
cpu MHz      : 2300.044
cache size   : 46080 KB
[...]

processor    : 1
vendor_id    : GenuineIntel
cpu family   : 6
model        : 79
model name   : Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
stepping     : 1
microcode    : 0xb00001d
cpu MHz      : 2300.044
cache size   : 46080 KB
[...]

$ free -m
              total        used        free      shared  buff/cache   available
Mem:           7982          62        7770           8         149        7701
Swap:             0           0           0
```

Your virtual machine can use two CPU cores and offers 7,982 MB of memory, compared to a single CPU core and 990 MB of memory before you increased the VM's size.

Cleaning up

Terminate the m4.large VM to stop paying for it:

1. Open the EC2 service from the main navigation, and select Instances from the submenu.
2. Select the running virtual machine by clicking the row in the table.
3. In the Actions menu, choose Instance State > Terminate.

3.5 Starting a virtual machine in another data center

AWS offers data centers all over the world. Take the following criteria into account when deciding which region to choose for your cloud infrastructure:

- *Latency*—Which region offers the shortest distance between your users and your infrastructure?
- *Compliance*—Are you allowed to store and process data in that country?
- *Service availability*—AWS does not offer all services in all regions. Are the services you are planning to use available in the region? Check out the service availability region table at http://mng.bz/0q40.
- *Costs*—Service costs vary by region. Which region is the most cost-effective region for your infrastructure?

Let's assume you have customers not just in the United States but in Australia as well. At the moment you are only operating EC2 instances in N. Virginia (US). Customers from Australia complain about long loading times when accessing your website. To make your Australian customers happy, you decide to launch an additional VM in Australia.

Changing a data center is simple. The Management Console always shows the current data center you're working in, on the right side of the main navigation menu. So far, you've worked in the data center N. Virginia (US), called us-east-1. To change the data center, click N. Virginia and select Sydney from the menu. Figure 3.19 shows how to jump to the data center in Sydney called ap-southeast-2.

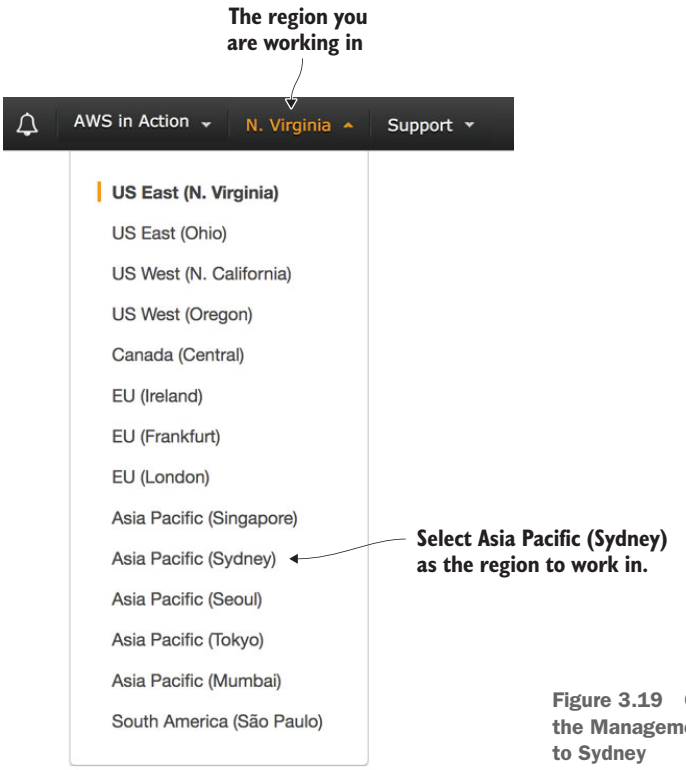


Figure 3.19 Changing the data center in the Management Console from N. Virginia to Sydney

AWS groups its data centers into these regions:

- US East, N. Virginia (us-east-1)
- US West, N. California (us-west-1)
- Canada, Central (ca-central-1)
- EU, Frankfurt (eu-central-1)
- EU, Paris (eu-west-3)
- Asia Pacific, Seoul (ap-northeast-2)
- Asia Pacific, Sydney (ap-southeast-2)
- South America, São Paulo (sa-east-1)
- US East, Ohio (us-east-2)
- US West, Oregon (us-west-2)
- EU, Ireland (eu-west-1)
- EU, London (eu-west-2)
- Asia Pacific, Tokyo (ap-northeast-1)
- Asia Pacific, Singapore (ap-southeast-1)
- Asia Pacific, Mumbai (ap-south-1)

You can specify the region for most AWS services. The regions are independent of each other; data isn't transferred between regions. Typically, a region is a collection of three or more data centers located in the same area. Those data centers are well connected to each other and offer the ability to build a highly available infrastructure, as you'll discover later in this book. Some AWS services, like CDN and the Domain Name System (DNS) service, act globally on top of these regions and even on top of some additional data centers.

After you change to the EC2 service in the Management Console, you may wonder why no key pair is listed in the EC2 overview. You created a key pair for SSH logins in the region N. Virginia (US). But the regions are independent, so you have to create a new key pair for the Sydney region. Follow these steps (see section 1.2 if you need more details):

1. Open the EC2 service from the main navigation, and select Key Pairs from the submenu.
2. Click Create Key Pair, and type in `sydney` as the key pair name.
3. Download and save the key pair.
4. Windows only: Open PuTTYgen, and select SSH-2 RSA under Type of Key to Generate. Click Load. Select the sydney.pem file, and click Open. Confirm the dialog box. Click Save Private Key.
5. Linux and macOS only: Change the access rights of the file sydney.pem by running `chmod 400 sydney.pem` in the terminal.

You're ready to start a virtual machine in the data center in Sydney. Follow these steps to do so:

1. Open the EC2 service from the main navigation menu, and select Instances from the submenu.
2. Click Launch Instance to start a wizard that will guide you through starting a new virtual machine.
3. Select the Amazon Linux AMI (HVM) machine image.
4. Choose t2.micro as the instance type, and click Review and Launch to go to the next screen.
5. Click Edit Security Groups to configure the firewall. Change the Security Group Name to `webserver` and the Description to `HTTP and SSH`. Add two rules: one of type SSH and another of type HTTP. Allow access to SSH and HTTP from anywhere by defining `0.0.0.0/0` as the source for both rules. Your firewall configuration should look like figure 3.20. Click Review and Launch.
6. Click Launch, and select `sydney` as the existing key pair to use for launching your VM.
7. Click View Instances to see the overview of virtual machines, and wait for your new VM to start.

You're finished! A virtual machine is running in a data center in Sydney. Let's proceed with installing a web server on it. To do so, you have to connect to your virtual machine via SSH. Grab the current public IP address of your virtual machine from its Details page.

Open a terminal, and type `ssh -i` *`$PathToKey`*`/sydney.pem ec2-user@$PublicIp`, replacing *`$PathToKey`* with the path to the key file sydney.pem that you downloaded, and *`$PublicIp`* with the public IP address from the details of your virtual machine. Answer Yes to the security alert about the authenticity of the new host.

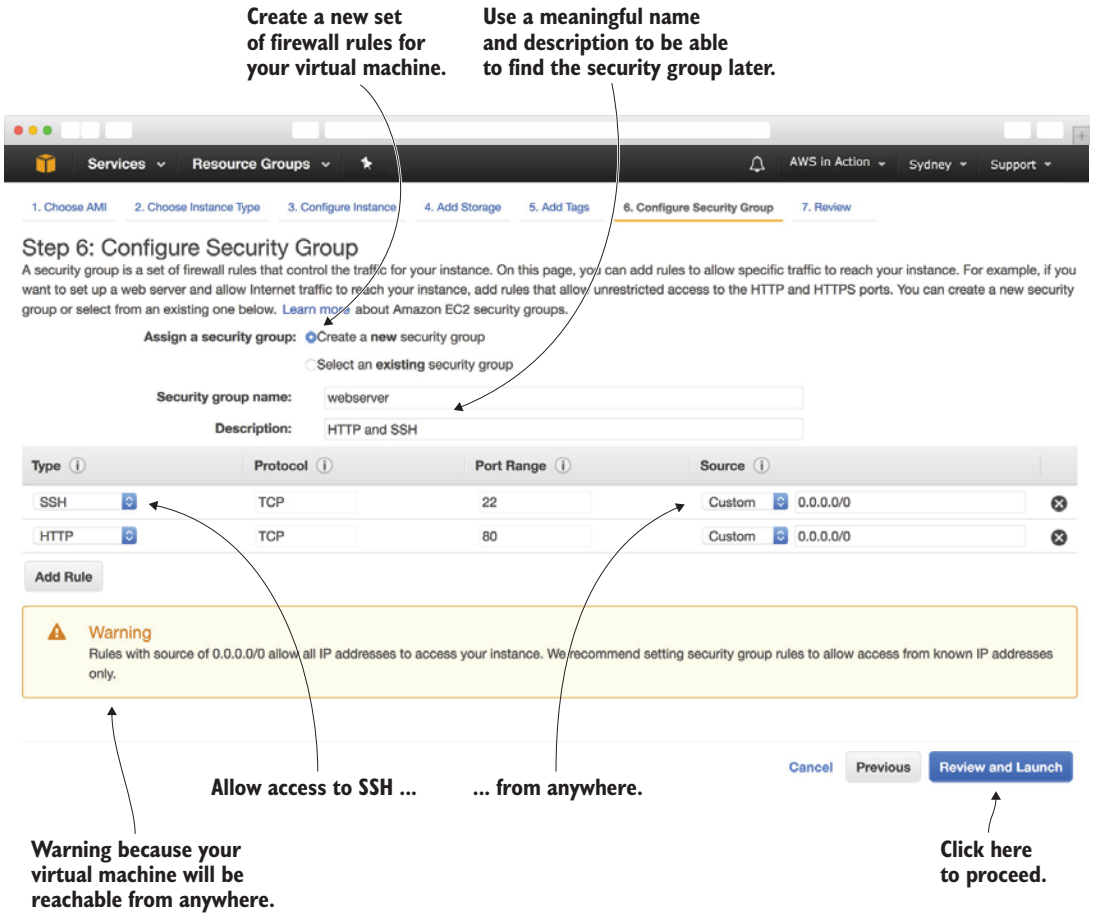


Figure 3.20 Configuring the firewall for a web server in Sydney

Windows

Find the sydney.ppk file you created after downloading the new key pair, and open it by double-clicking. The PuTTY Pageant should appear in the taskbar as an icon. Next, start PuTTY and connect to the public IP address you got from the Details page of your virtual machine. Answer Yes to the security alert regarding the authenticity of the new host, and type in `ec2-user` as login name. Press Enter.

To serve your website to your Australian customers, connect to the EC2 instance via SSH and install a web server by executing `sudo yum install httpd -y`. To start the web server, type `sudo service httpd start` and press Return to execute the command. Your web browser should show a placeholder site if you open http://*$PublicIp*, with *$PublicIp* replaced by the public IP address of your virtual machine.

NOTE You're using two different operating systems in this chapter. You started with a VM based on Ubuntu at the beginning of the chapter. Now you're using Amazon Linux, a distribution based on Red Hat Enterprise Linux. That's why you have to execute different commands to install software. Ubuntu uses `apt-get` and Amazon Linux is using `yum`.

Next, you'll attach a fixed public IP address to the virtual machine.

3.6 Allocating a public IP address

You've already launched some virtual machines while reading this book. Each VM was connected to a public IP address automatically. But every time you launched or stopped a VM, the public IP address changed. If you want to host an application under a fixed IP address, this won't work. AWS offers a service called *Elastic IPs* for allocating fixed public IP addresses. You can allocate a public IP address and associate it with an EC2 instance by following these steps in figure 3.21:

1. Open the Management Console, and go to the EC2 service.
2. Choose Elastic IPs from the submenu. You'll see an overview of public IP addresses.
3. Allocate a public IP address by clicking Allocate New Address.
4. Confirm by clicking on Allocate.
5. Your fixed public IP address is shown. Click Close to go back to the overview.

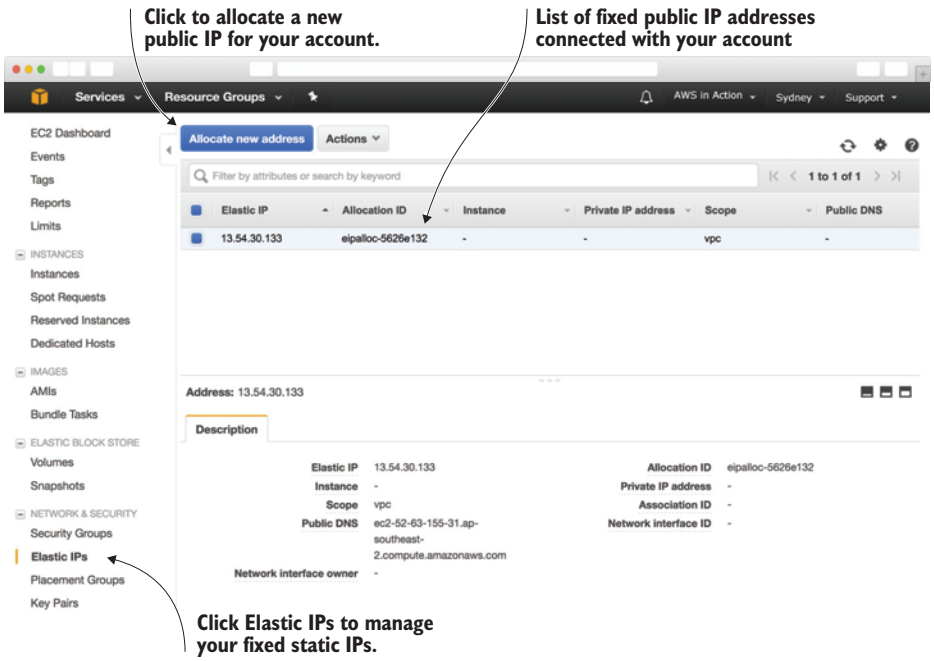


Figure 3.21 Overview of public IP addresses connected to your account in the current region

Now you can associate the public IP address with a virtual machine of your choice:

1. Select your public IP address, and choose Associate Address from the Actions menu. A dialog similar to figure 3.22 appears.
2. Select Instance as the Resource Type.
3. Enter your EC2 instance's ID in the Instance field. There is only a single virtual machine running at the moment, so only one option is available.
4. Only one Private IP is available for your virtual machine. Select it.
5. Click Associate to finish the process.

Your virtual machine is now accessible through the public IP address you allocated at the beginning of this section. Point your browser to this IP address, and you should see the placeholder page as you did in section 3.5.

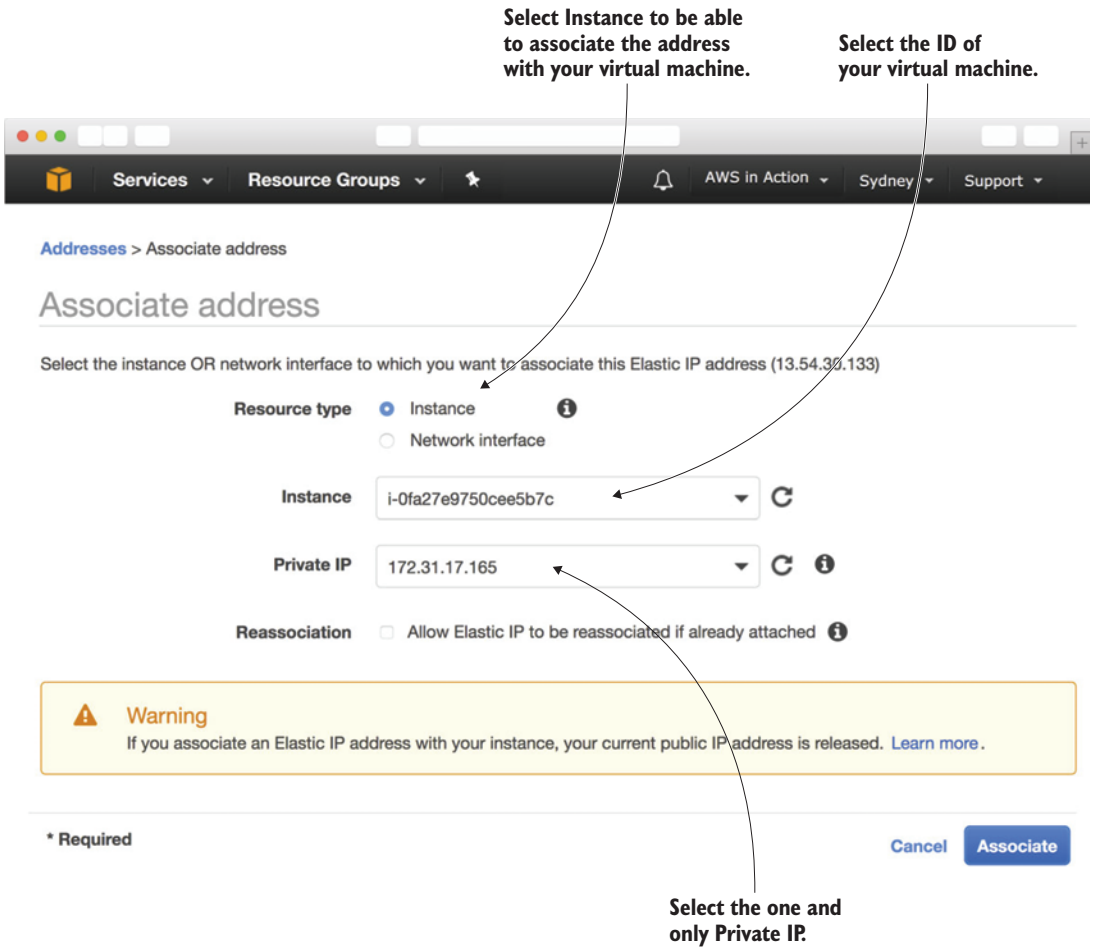


Figure 3.22 Associating a public IP address with your EC2 instance

Allocating a public IP address can be useful if you want to make sure the endpoint to your application doesn't change, even if you have to replace the virtual machine behind the scenes. For example, assume that virtual machine A is running and has an associated Elastic IP. The following steps let you replace the virtual machine with a new one without changing the public IP address:

1. Start a new virtual machine B to replace running virtual machine A.
2. Install and start applications as well as all dependencies on virtual machine B.
3. Disassociate the Elastic IP from virtual machine A, and associate it with virtual machine B.

Requests using the Elastic IP address will now be routed to virtual machine B, with a short interruption while moving the Elastic IP. You can also connect multiple public IP addresses with a virtual machine by using multiple network interfaces, as described in the next section. This can be useful if you need to host different applications running on the same port, or if you want to use a unique fixed public IP address for different websites.

> **WARNING** IPv4 addresses are scarce. To prevent stockpiling Elastic IP addresses, AWS will charge you for Elastic IP addresses that aren't associated with a virtual machine. You'll clean up the allocated IP address at the end of the next section.

3.7 Adding an additional network interface to a virtual machine

In addition to managing public IP addresses, you can control your virtual machine's network interfaces. It is possible to add multiple network interfaces to a VM and control the private and public IP addresses associated with those network interfaces.

Here are some typical use cases for EC2 instances with multiple network interfaces:

- Your web server needs to answer requests by using multiple TLS/SSL certificates, and you can't use the Server Name Indication (SNI) extension due to legacy clients.
- You want to create a management network separated from the application network, and therefore your EC2 instance needs to be accessible from two networks. Figure 3.23 illustrates an example.
- Your application requires or recommends the use of multiple network interfaces (for example, network and security appliances).

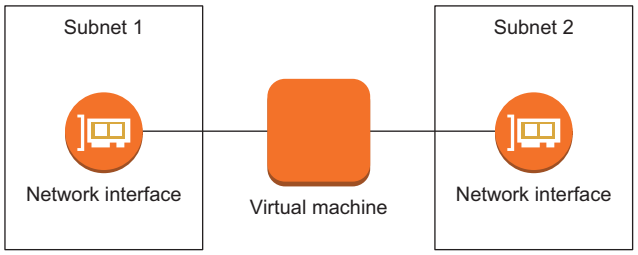


Figure 3.23 A virtual machine with two network interfaces in two different subnets

You use an additional network interface to connect a second public IP address to your EC2 instance. Follow these steps to create an additional networking interface for your virtual machine:

1. Open the Management Console, and go to the EC2 service.
2. Select Network Interfaces from the submenu.
3. The default network interface of your virtual machine is shown in the list. Note the subnet ID of the network interface.
4. Click Create Network Interface. A dialog like the one shown in figure 3.24 appears.
5. Enter `2nd interface` as the description.
6. Choose the subnet you noted down in step 3.
7. Leave Private IP Address empty. A private IP will be assigned to the network interface automatically.
8. Select the Security Groups that have *webserver* in their description.
9. Click Yes, Create.

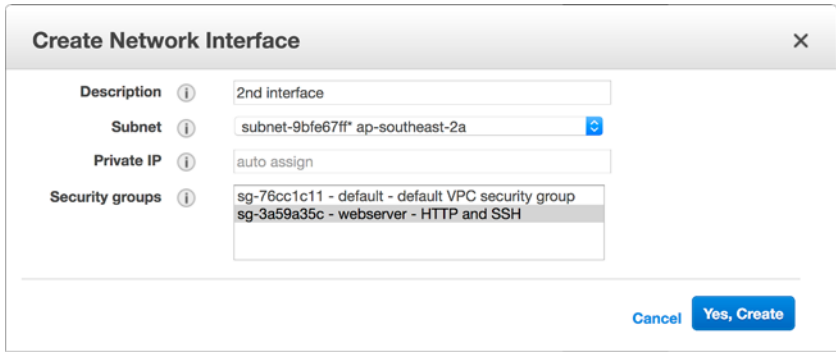


Figure 3.24 Creating an additional networking interface for your virtual machine

After the new network interface's state changes to Available, you can attach it to your virtual machine. Select the new 2nd interface network interface, and choose Attach from the menu. A dialog opens like shown in figure 3.25. Choose the only available Instance ID, and click Attach.

You've attached an additional networking interface to your virtual machine. Next, you'll connect an additional public IP address to the additional networking interface. To do so, note down the network interface ID of the 2nd interface shown in the overview—`eni-b847f4c5` in our example—and follow these steps:

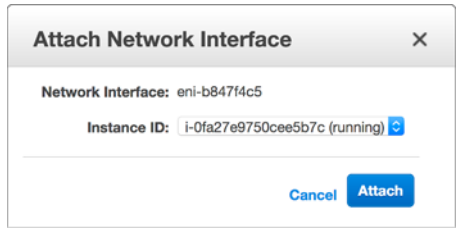


Figure 3.25 Attaching an additional networking interface to your virtual machine

1. Open the AWS Management Console, and go to the EC2 service.
2. Choose Elastic IPs from the submenu.
3. Click Allocate New Address to allocate a new public IP address, as you did in section 3.6.
4. Select your public IP address, and choose Associate Address from the Actions menu. A dialog similar to figure 3.26 appears.
5. Select Network interface as the Resource Type.
6. Enter your 2nd interface's ID in the Network Interface field.
7. Select the only available Private IP for your network interface.
8. Click Associate to finish the process.

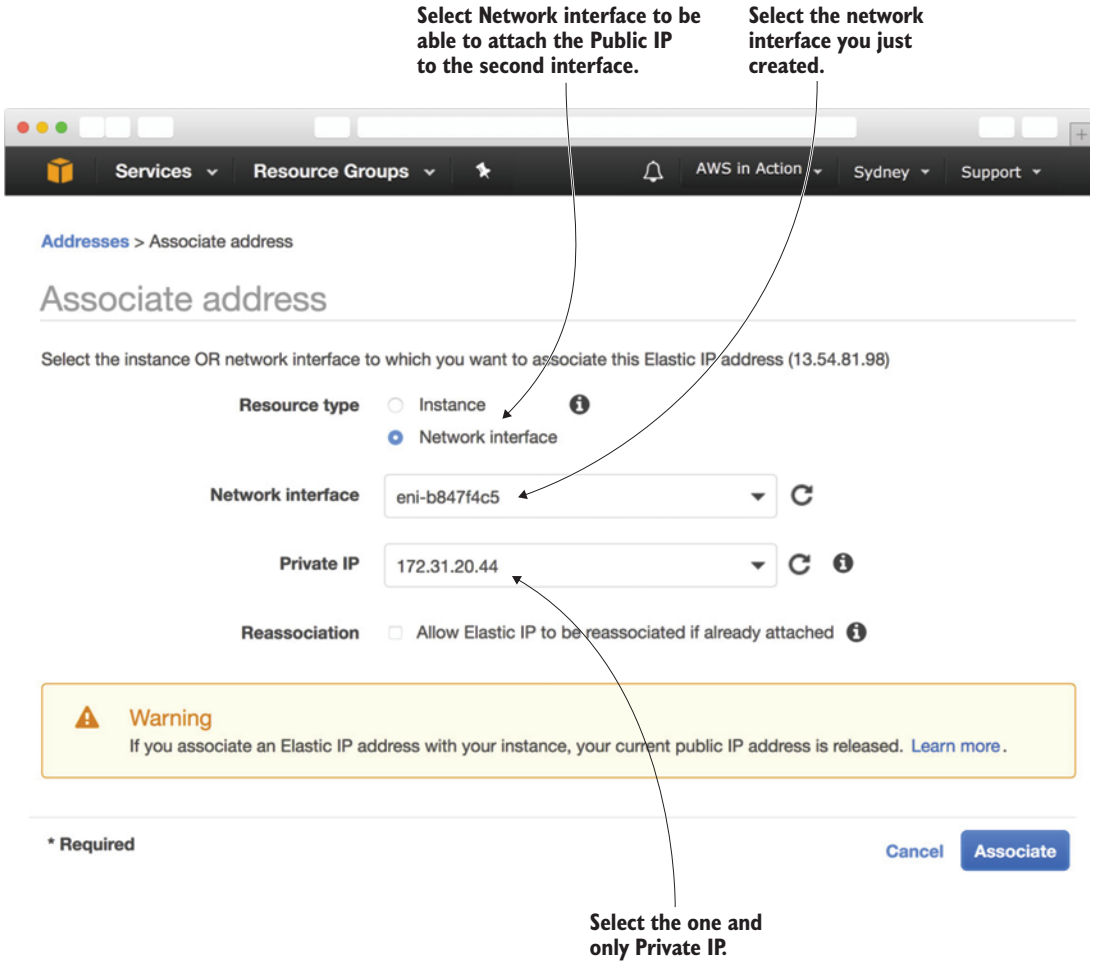


Figure 3.26 Associating a public IP address with the additional networking interface

Your virtual machine is now reachable under two different public IP addresses. This enables you to serve two different websites, depending on the public IP address. You need to configure the web server to answer requests depending on the public IP address.

If you connect to your virtual machine via SSH and insert `ifconfig` into the terminal, you can see your new networking interface attached to the virtual machine, as shown in the following output after running the `ifconfig` command:

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 02:06:EE:59:F7:65
          inet addr:172.31.17.165  Bcast:172.31.31.255  Mask:255.255.240.0
          inet6 addr: fe80::6:eeff:fe59:f765/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:3973 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2648 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5104781 (4.8 MiB)  TX bytes:167733 (163.8 KiB)

eth1      Link encap:Ethernet  HWaddr 02:03:FA:95:9B:BB
          inet addr:172.31.20.44  Bcast:172.31.31.255  Mask:255.255.240.0
          inet6 addr: fe80::3:faff:fe95:9bbb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:9001  Metric:1
          RX packets:84 errors:0 dropped:0 overruns:0 frame:0
          TX packets:87 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9997 (9.7 KiB)  TX bytes:11957 (11.6 KiB)
[...]
```

Each network interface is connected to a private and a public IP address. You'll need to configure the web server to deliver different websites depending on the IP address. Your virtual machine doesn't know anything about its public IP address, but you can distinguish the requests based on the private IP address.

First you need two websites. Run the following commands via SSH on your virtual machine in Sydney to download two simple placeholder websites:

```
$ sudo -s
$ mkdir /var/www/html/a
$ wget -P /var/www/html/a https://raw.githubusercontent.com/AWSinAction/\
➥ code2/master/chapter03/a/index.html
$ mkdir /var/www/html/b
$ wget -P /var/www/html/b https://raw.githubusercontent.com/AWSinAction/\
➥ code2/master/chapter03/b/index.html
```

Next you need to configure the web server to deliver the websites depending on which IP address is called. To do so, add a file named `a.conf` under /etc/httpd/conf.d with the following content. Change the IP address from 172.31.x.x to the IP address from the `ifconfig` output for the networking interface eth0:

```
<VirtualHost 172.31.x.x:80>
  DocumentRoot /var/www/html/a
</VirtualHost>
```

Repeat the same process for a configuration file named `b.conf` under /etc/httpd/conf.d with the following content. Change the IP address from 172.31.y.y to the IP address from the `ifconfig` output for the networking interface eth1:

```
<VirtualHost 172.31.y.y:80>
  DocumentRoot /var/www/html/b
</VirtualHost>
```

To activate the new web server configuration, execute `sudo service httpd restart` via SSH. Change to the Elastic IP overview in the Management Console. Copy both public IP addresses, and open them with your web browser. You should get the answer "Hello A!" or "Hello B!" depending on the public IP address you're calling. Thus you can deliver two different websites, depending on which public IP address the user is calling. Congrats—you're finished!

Cleaning up

It's time to clean up your setup:

1. Terminate the virtual machine.
2. Go to Networking Interfaces, and select and delete the networking interface.
3. Change to Elastic IPs, and select and release the two public IP addresses by clicking Release Addresses from the Actions menu.
4. Go to Key Pairs, and delete the sydney key pair you created.
5. Go to Security Groups, and delete the webserver security group you created.

That's it. Everything is cleaned up, and you're ready for the next section.

NOTE You switched to the AWS region in Sydney earlier. Now you need to switch back to the region US East (N. Virginia). You can do so by selecting US East (N. Virginia) from the region chooser in the main navigation menu of the Management Console.

3.8 Optimizing costs for virtual machines

Usually you launch virtual machines *on demand* in the cloud to gain maximum flexibility. AWS calls them on-demand instances, because you can start and stop VMs on-demand, whenever you like, and you're billed for every second or hour the machine is running.

There are two options to reduce EC2 costs: *spot instances* or *reserved instances*. Both help to reduce costs but decrease your flexibility. With a spot instance, you bid for unused capacity in an AWS data center; the price is based on supply and demand. You can use reserved instances if you need a virtual machine for a year or longer; you agree to pay for the given time frame and receive a discount in advance. Table 3.2 shows the differences between these options.

Billing unit: Seconds

Most EC2 instances running Linux (such as Amazon Linux or Ubuntu) are billed per second. The minimum charge per instance is 60 seconds. For example, if you terminate a newly launched instance after 30 seconds, you have to pay for 60 seconds. But if you terminate an instance after 61 seconds, you pay exactly for 61 seconds.

An EC2 instance running Microsoft Windows or a Linux distribution with an extra hourly charge (such as Red Hat Enterprise Linux or SUSE Linux Enterprise Server) is not billed per second, but per hour. A minimum charge of one hour applies. The same is true for EC2 instances with an extra hourly charge launched from the AWS Marketplace.

Table 3.2 Differences between on-demand, reserved, and spot virtual machines

	On-demand	Reserved	Spot
Price	High	Medium	Low
Flexibility	High	Low	Medium
Reliability	Medium	High	Low
	Dynamic workloads (for example, for a news site) or proof-of-concept	Predictable and static workloads (for example, for a business application)	Batch workloads (for example, for data analytics, media encoding, ...)

3.8.1 Reserve virtual machines

Reserving a virtual machine means committing to using a specific VM in a specific data center. You have to pay for a reserved VM whether or not it's running. In return, you benefit from a price reduction of up to 60%. On AWS, you can choose one of the following options if you want to reserve a virtual machine:

- No Upfront, 1-year term
- Partial Upfront, 1- or 3-year term
- All Upfront, 1- or 3-year term

Table 3.3 shows what this means for an m4.large virtual machine with two virtual CPUs and 8 GB of memory.

WARNING Buying a reservation will incur costs for 1 or 3 years. That's why we did not add an example for this section.

Table 3.3 Potential cost savings for a virtual machine with instance type m4.large

	Monthly cost	Upfront cost	Effective monthly cost	Savings vs. on-demand
On-demand	$73.20 USD	$0.00 USD	$73.20 USD	N/A
No Upfront, 1-year term, standard	$45.38 USD	$0.00 USD	$45.38 USD	38%

Table 3.3 Potential cost savings for a virtual machine with instance type m4.large *(continued)*

	Monthly cost	Upfront cost	Effective monthly cost	Savings vs. on-demand
Partial Upfront, 1-year term, standard	$21.96 USD	$258.00 USD	$43.46 USD	40%
All Upfront, 1-year term, standard	$0.00 USD	$507.00 USD	$42.25 USD	42%
No Upfront, 3-year term, standard	$31.47 USD	$0.00 USD	$31.47 USD	57%
Partial Upfront, 3-year term, standard	$14.64 USD	$526.00 USD	$29.25 USD	60%
All Upfront, 3-year term, standard	$0.00 USD	$988.00 USD	$27.44 USD	63%

You can trade cost reductions against flexibility by reserving virtual machines on AWS. There are different options offering different levels of flexibility when buying a reserved instance:

- Reserved instances, either with or without capacity reservation
- Standard or convertible reserved instances

We'll explain these in more detail in the following sections. It is possible to buy reservations for specific time frames, called Scheduled Reserved Instances. For example, you can make a reservation for every workday from 9 a.m. to 5 p.m.

RESERVED INSTANCES WITH CAPACITY RESERVATION

If you own a reservation for a virtual machine (a *reserved instance*) in a specific availability zone, the capacity for this virtual machine is reserved for you in the public cloud. Why is this important? Suppose demand increases for VMs in a particular data center, perhaps because another data center broke down and many AWS customers have to launch new virtual machines to replace their broken ones. In this rare case, the orders for on-demand VMs will pile up, and it may become nearly impossible to start a new VM in that data center. If you plan to build a highly available setup across multiple data centers, you should also think about reserving the minimum capacity you'll need to keep your applications running. The downside of a capacity reservation is low flexibility. You commit to pay for a VM of a specific type within a specific data center for one or three years.

RESERVED INSTANCES WITHOUT CAPACITY RESERVATION

If you choose not to reserve capacity for your virtual machine, you can profit from reduced hourly prices and more flexibility. A reserved instance without capacity reservation is valid for VMs in a whole region, no matter which specific data center is used. The reservation is applicable to all instance sizes of an instance family as well. For example, if you buy a reservation for a m4.xlarge machine, you can also run two m4.large machines making use of the reservation as well.

Modifying reservations

It is possible to modify a reservation without additional charges. Doing so allows you to adapt your reservations to changes in your workload over time. You can:

- Toggle capacity reservation.
- Modify which data center your reservation is in.
- Split or merge a reservation. For example, you can merge two t2.small reservations into a t2.medium reservation.

STANDARD OR CONVERTIBLE?

When buying a reservation for three years, you have the option to chose between standard or convertible offerings. A *standard reservation* limits you to a specific instance family (such as m4). A *convertible reservation* can be exchanged for another reservation, for example to change to a different instance family. Convertible reservations are more expensive than standard reservations but offer higher flexibility.

Being able to switch to another instance family might be valuable because AWS might introduce a new instance family with lower prices and higher resources in the future. Or perhaps your workload pattern changes and you want to switch the instance family, for example from a general-purpose family to a one that's optimized for computing.

We recommend that you start with on-demand machines and switch to a mix of on-demand and reserved instances later. Depending on your high availability requirements, you can choose between reservations with and without capacity reservation.

3.8.2 Bidding on unused virtual machines

In addition to reserved virtual machines, there is another option for reducing costs: *spot instances*. With a spot instance, you bid for unused capacity in the AWS cloud. A *spot market* is a market where standardized products are traded for immediate delivery. The price of the products on the market depends on supply and demand. On the AWS spot market, the products being traded are VMs, and they're delivered by starting a virtual machine.

Figure 3.27 shows the price chart for a specific instance type for a virtual machine. If the current spot price is lower than your maximum bid for a specific virtual machine VM in a specific data center, your spot request will be fulfilled, and a virtual machine will start, billed at the spot price. If the current spot price exceeds your bid, your VM will be terminated (not stopped) by AWS after two minutes.

The spot price can be more or less flexible depending on the size of the VMs and the data center. We've seen everything from a spot price that was only 10% of the on-demand price to a spot price that was greater than the on-demand price. As soon as the spot price exceeds your bid, your EC2 instance will be terminated within two minutes. You shouldn't use spot instances for tasks like web or mail servers, but you can use them to run asynchronous tasks like analyzing data or encoding media assets. You

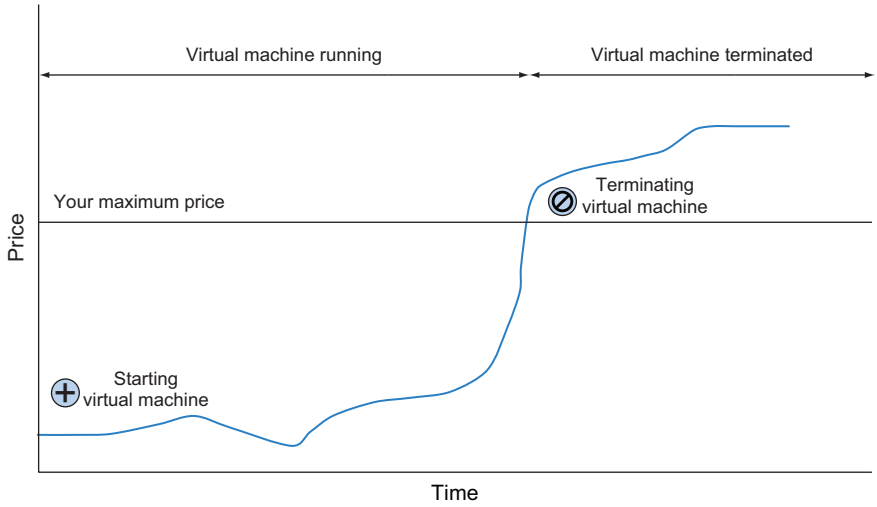


Figure 3.27 Functionality of the spot market for virtual machines

can even use a spot instance to check for broken links on your website, as you did in section 3.1, because this isn't a time-critical task.

Let's start a new virtual machine that uses the price reductions of the spot market. First you have to place your order on the spot market; figure 3.28 shows the starting point for requesting a spot instance. You get there by choosing the EC2 service from the main navigation menu and selecting Spot Requests from the submenu. Click on Pricing History. A dialog appears showing the spot prices for virtual machines; historical prices are available for the different server sizes and different data centers.

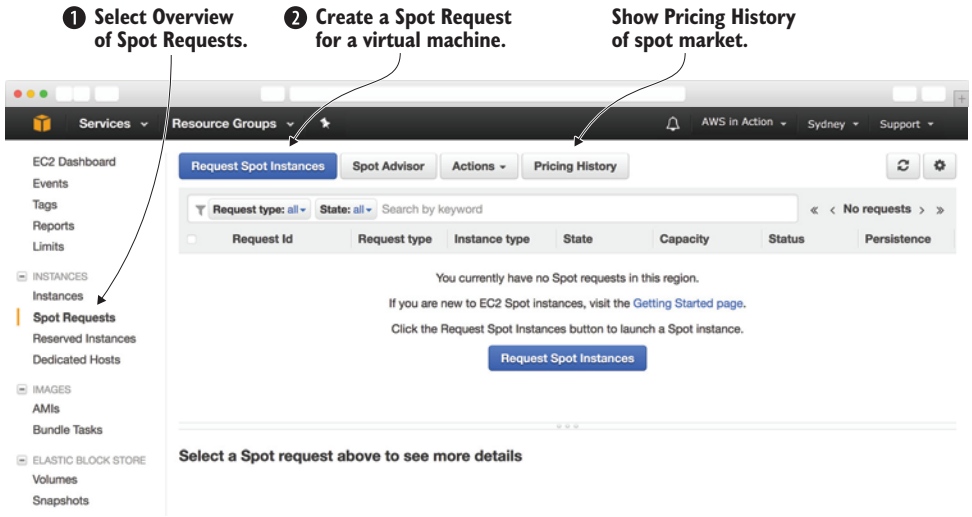


Figure 3.28 Requesting a spot instance

Click Request Spot Instances to start the wizard that guides you through the process of requesting a spot instance.

> **WARNING** Starting a virtual machine with instance type m3.medium via spot request incurs charges. The maximum price (bid) is $0.093 in the following example.

The steps in figure 3.29 are necessary to start the smallest available virtual machine available on the spot market:

1. You can choose between three request types. *Request* will request a one-time virtual machine. *Request and Maintain* will create a fleet of VMs and keep them running at the lowest possible price. Or *Reserve for Duration* will request a virtual machine with a guaranteed lifetime of 1 to 6 hours. Choose *Request* to request a single virtual machine.
2. Request a target capacity of one EC2 instance.
3. Select the Ubuntu 16.04 LTS AMI.
4. Choose m3.medium, the smallest available instance type for spot instances.
5. Keep the defaults for allocation strategy and network.
6. Set your own maximum price. In this example, we're using the on-demand price of $0.093 USD as a maximum. Prices are subject to change. Get the latest prices from https://aws.amazon.com/ec2/pricing/on-demand/, the Amazon EC2 pricing page.
7. Click the Next button to proceed with the next step.

The next step, as shown in figure 3.30, is to configure the details of your virtual machine and the spot request. Set the following parameters:

1. Keep the default storage settings. A network-attached volume with 8 GB will be enough to run the link checker.
2. Keep the defaults for the EC2 instance as well, expect for the Instance tags. Add a tag with key `Name` and value `spotinstance` so that you can identify your virtual machine later.
3. Select your Key Pair named `mykey` and keep the default for IAM instance profile and IAM fleet role.
4. Select the `ssh-only` security group, which you created at the beginning of the chapter to allow incoming SSH traffic to your virtual machine.
5. Keep the default for the Public IP as well as the request validity.
6. Click the Review button to proceed to the next step.

The last step of the wizard shows a summary of your spot request. Click the Launch button to create the request.

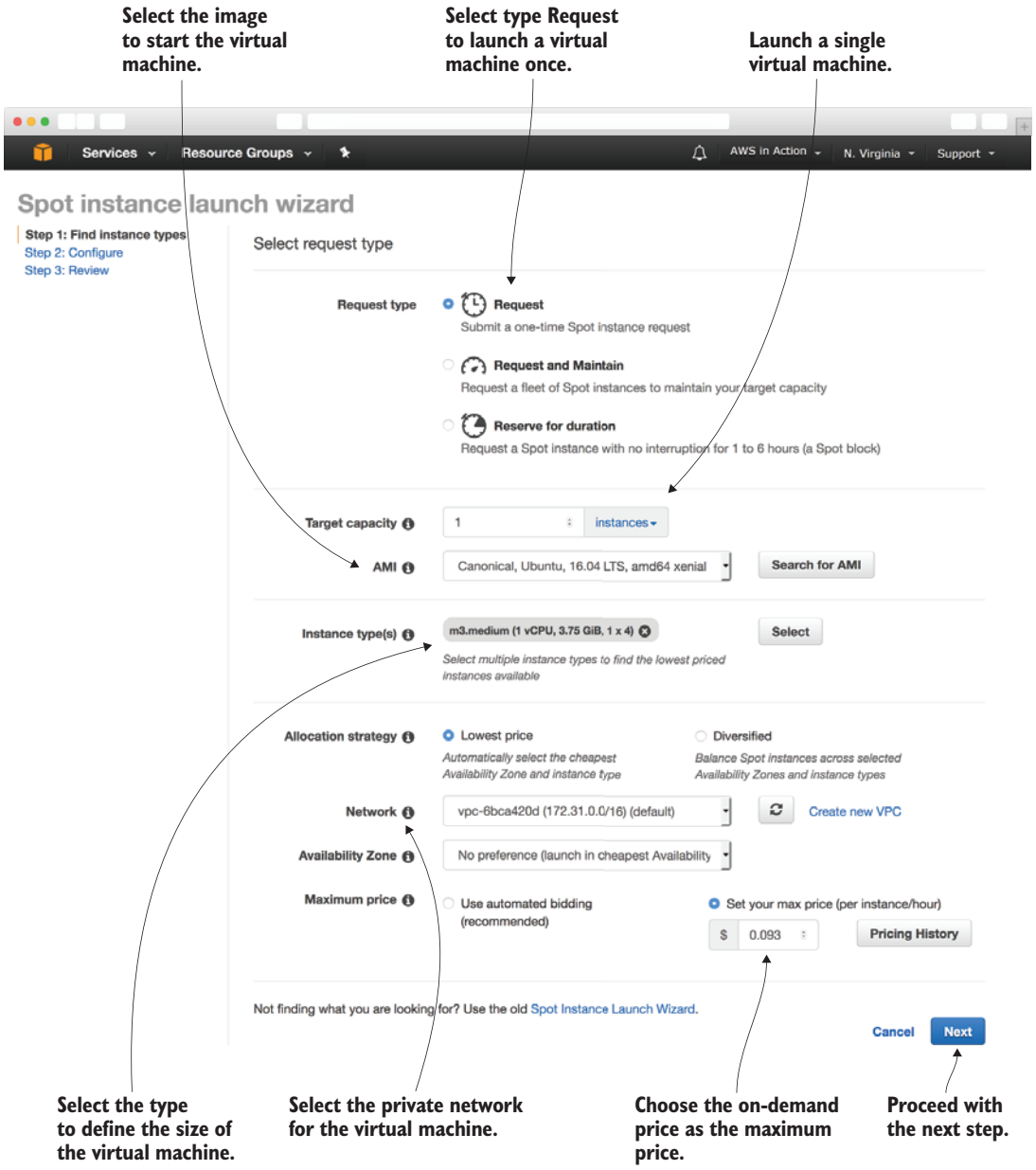


Figure 3.29 Requesting a spot instance (step 1)

Figure 3.30 Requesting a spot instance (step 2)

After you finish the wizard, your request for a virtual machine is placed on the market. Your spot request appears within the overview shown in figure 3.31. It may take several minutes for your request to be fulfilled. Look at the Status field for your request: because the spot market is unpredictable, it's possible that your request could fail. If this happens, repeat the process to create another request, and choose another instance type.

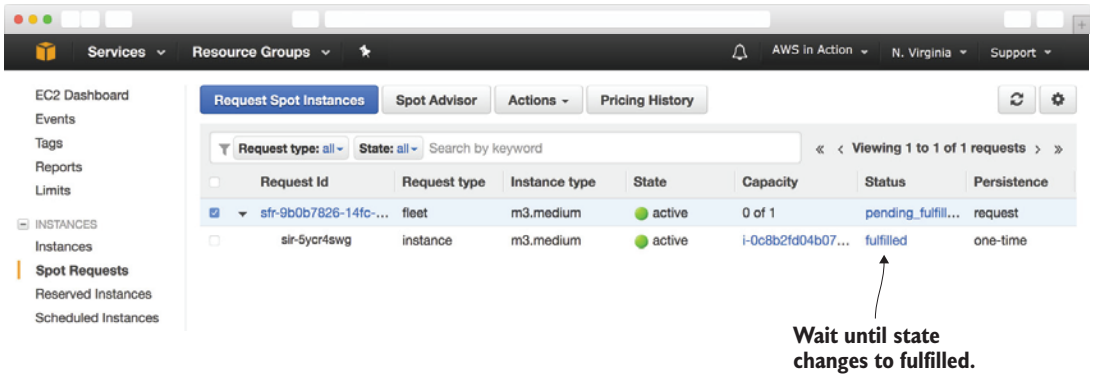


Figure 3.31 Waiting for the spot request to be fulfilled and the virtual machine to start

When the state of your requests flips to fulfilled, a virtual machine is started. You can look at it after switching to Instances via the submenu; you'll find a running or starting instance listed in the overview of virtual machines. You've successfully started a virtual machine that is billed as spot instance! You can now use SSH to connect to the VM and run the link checker as described in section 3.1.3.

Cleaning up

Terminate the m3.medium VM to stop paying for it:

1. Open the EC2 service from the main navigation menu, and select Spot Requests from the submenu.
2. Select your spot request by clicking the first row in the table.
3. In the Actions menu, choose Cancel Spot request.
4. Make sure the option Terminate Instances is selected, click OK to confirm.

Summary

- You can choose an OS when starting a virtual machine.
- Using logs and metrics can help you to monitor and debug your VM.
- If you want to change the size of your VM, you can change the number of CPUs, as well as the amount of memory and storage.

- You can start VMs in different regions all over the world to provide low latency for your users.
- Allocating and associating a public IP address to your virtual machine gives you the flexibility to replace a VM without changing the public IP address.
- You can save on costs by reserving virtual machines or bidding for unused capacity on the virtual machine spot market.

4 Programming your infrastructure: The command-line, SDKs, and CloudFormation

This chapter covers

- Understanding the idea of infrastructure as code
- Using the CLI to start a virtual machine
- Using the JavaScript SDK for Node.js to start a virtual machine
- Using CloudFormation to start a virtual machine

Imagine that you want to provide room lighting as a service. To switch off the lights in a room using software, you need a hardware device like a relay connected to the light circuit. This hardware device must have some kind of interface that lets you send commands via software. With a relay and an interface, you can offer room lighting as a service.

This also applies to providing VMs as a service. VMs are software, but if you want to be able to start them remotely, you still need hardware that can handle and fulfill

your request. AWS provides an *application programming interface* (API) that can control every part of AWS over HTTP. Calling the HTTP API is very low-level and requires a lot of repetitive work, like authentication, data (de)serialization, and so on. That's why AWS offers tools on top of the HTTP API that are easier to use. Those tools are:

- *Command-line interface (CLI)*—With one of the CLIs, you can make calls to the AWS API from your terminal.
- *Software development kit (SDK)*—SDKs, available for most programming languages, make it easy to call the AWS API from your programming language of choice.
- *AWS CloudFormation*—Templates are used to describe the state of the infrastructure. AWS CloudFormation translates these templates into API calls.

Not all examples are covered by the Free Tier

The examples in this chapter are *not* all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

On AWS, everything can be controlled via an API. You interact with AWS by making calls to the REST API using the HTTPS protocol, as figure 4.1 illustrates. Everything is available through the API. You can start a virtual machine with a single API call, create 1 TB of storage, or start a Hadoop cluster over the API. By everything, we really mean *everything*. You'll need some time to understand the consequences. By the time you finish this book, you'll ask why the world wasn't always this easy.

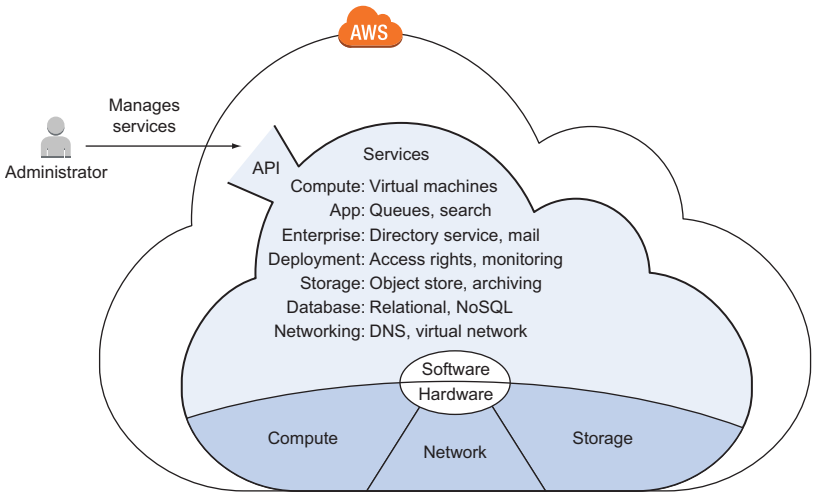
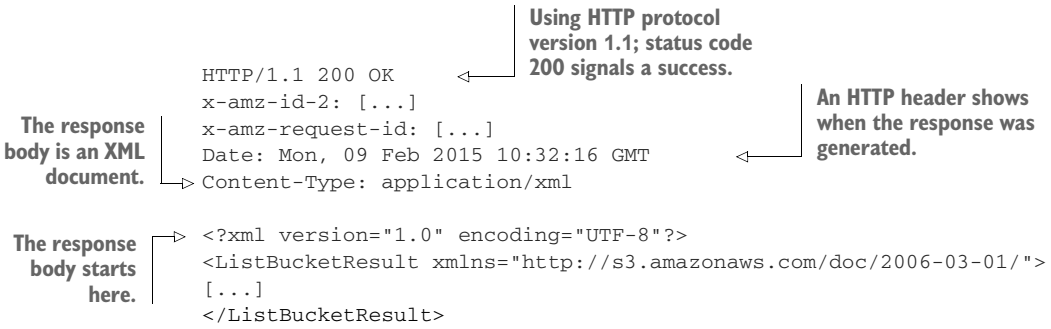


Figure 4.1 Interacting with AWS by making calls to the REST API

Let's look at how the API works. Imagine you uploaded a few files to the object store S3 (you will learn about S3 in chapter 8). Now you want to list all the files in the `S3` object store to check if the upload was successful. Using the raw HTTP API, you send a `GET` request to the API endpoint using the HTTP protocol:

The HTTP response will look like this:



Calling the API directly using plain HTTPS requests is inconvenient. The easy way to talk to AWS is by using the CLI or SDKs, as you learn in this chapter. But the API is the foundation of all those tools.

4.1 Infrastructure as Code

Infrastructure as Code is the idea of using a high-level programming language to control infrastructures. Infrastructure can be any AWS resource, like a network topology, a load balancer, a DNS entry, and so on. In software development, tools like automated tests, code repositories, and build servers increase the quality of software engineering. If your infrastructure is code, you can apply these tools to your infrastructure and improve its quality.

> **WARNING** Don't mix up the terms *Infrastructure as Code* and *Infrastructure as a Service* (IaaS)! IaaS means renting virtual machines, storage, and network with a pay-per-use pricing model.

4.1.1 Automation and the DevOps movement

The *DevOps movement* aims to bring software development and operations together. This usually is accomplished in one of two ways:

- *Using mixed teams with members from both operations and development.* Developers become responsible for operational tasks like being on-call. Operators are involved from the beginning of the software development cycle, which helps make the software easier to operate.
- *Introducing a new role that closes the gap between developers and operators.* This role communicates a lot with both developers and operators, and cares about all topics that touch both worlds.

The goal is to develop and deliver software to the customer rapidly without a negative impact on quality. Communication and collaboration between development and operations are therefore necessary.

The trend toward automation has helped DevOps culture bloom, as it codifies the cooperation between development and operations. You can only do multiple deployments per day if you automate the whole process. If you commit changes to the repository, the source code is automatically built and tested against your automated tests. If the build passes the tests, it's automatically installed in your testing environment. This triggers some acceptance tests. After those tests have been passed, the change is propagated into production. But this isn't the end of the process; now you need to carefully monitor your system and analyze the logs in real time to ensure that the change was successful.

If your infrastructure is automated, you can spawn a new system for every change introduced to the code repository and run the acceptance tests isolated from other changes that were pushed to the repository at the same time. Whenever a change is made to the code, a new system is created (virtual machine, databases, networks, and so on) to run the change in isolation.

4.1.2 Inventing an infrastructure language: JIML

For the purposes of learning infrastructure as code in detail, let's invent a new language to describe infrastructure: JSON Infrastructure Markup Language (JIML). Figure 4.2 shows the infrastructure that will be created in the end.

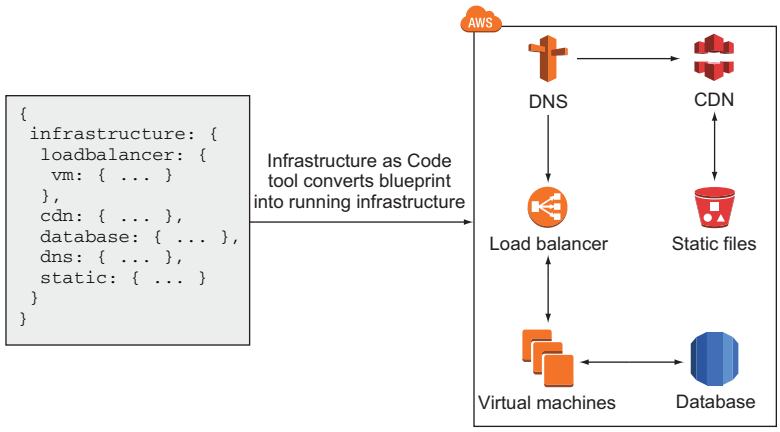


Figure 4.2 From JIML blueprint to infrastructure: infrastructure automation

The infrastructure consists of the following:

- Load balancer (LB)
- Virtual machines (VMs)
- Database (DB)
- DNS entry
- Content delivery network (CDN)
- Bucket for static files

To reduce issues with syntax, let's say JIML is based on JSON. The following JIML code creates the infrastructure shown in figure 4.2. The `$` indicates a reference to an ID.

Listing 4.1 Infrastructure description in JIML

```
{
  "region": "us-east-1",
  "resources": [{
    "type": "loadbalancer",
    "id": "LB",
    "config": {
      "virtualmachines": 2,
      "virtualmachine": {
        "cpu": 2,
        "ram": 4,
        "os": "ubuntu"
      }
    },
    "waitFor": "$DB"
  }, {
    "type": "cdn",
    "id": "CDN",
    "config": {
      "defaultSource": "$LB",
      "sources": [{
        "path": "/static/*",
        "source": "$BUCKET"
      }]
    }
  }, {
    "type": "database",
    "id": "DB",
    "config": {
      "password": "***",
      "engine": "MySQL"
    }
  }, {
    "type": "dns",
    "config": {
      "from": "www.mydomain.com",
      "to": "$CDN"
    }
  }, {
    "type": "bucket",
    "id": "BUCKET"
  }]
}
```

A load balancer is needed.

Need two VMs.

VMs are Ubuntu Linux (4 GB memory, 2 cores).

LB can only be created if the database is ready.

A CDN is used that caches requests to the LB or fetches static assets (images, CSS files, ...) from a bucket.

Data is stored within a MySQL database.

A DNS entry points to the CDN.

A bucket is used to store static assets (images, CSS files, ...).

How can we turn this JSON into AWS API calls?

1. Parse the JSON input.
2. The JIML tool creates a dependency graph by connecting the resources with their dependencies.
3. The JIML tool traverses the dependency graph from the bottom (leaves) to the top (root) and a linear flow of commands. The commands are expressed in a pseudo language.
4. The commands in pseudo language are translated into AWS API calls by the JIML runtime.

The AWS API calls have to be made based on the resources defined in the blueprint. In particular, it is necessary to send the AWS API calls in the correct order. Let's look at the dependency graph created by the JIML tool, shown in figure 4.3.

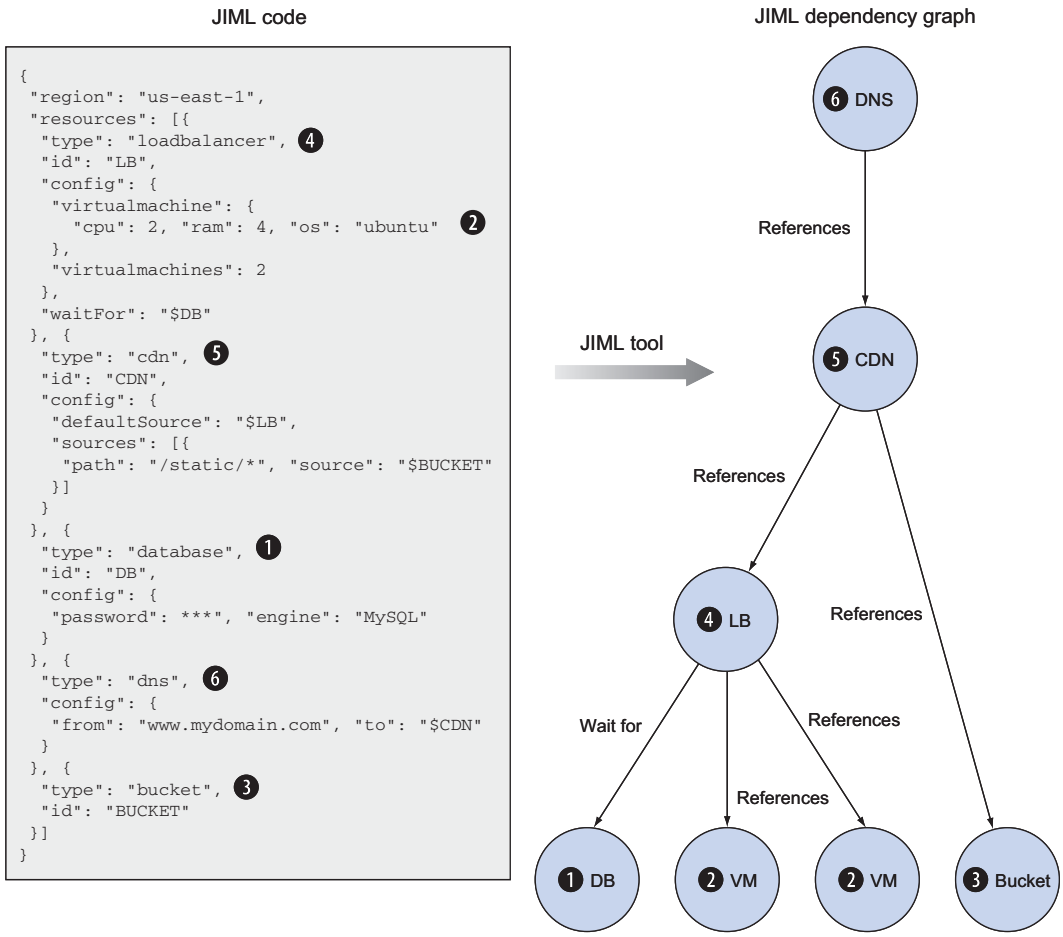


Figure 4.3 The JIML tool figures out the order in which resources need to be created.

You traverse the dependency graph in figure 4.3 from bottom to top and from left to right. The nodes at the bottom have no children: `DB` ❶, `virtualmachine` ❷, and `bucket` ❸. Nodes without children have no dependencies. The `LB` ❹ node depends on the `DB` node and the two `virtualmachine` nodes. The `CDN` ❺ node depends on the `LB` node and the `bucket` ❸ node. Finally, the `DNS` ❻ node depends on the `CDN` node.

The JIML tool turns the dependency graph into a linear flow of commands using pseudo language. The pseudo language represents the steps needed to create all the resources in the correct order. The nodes are easy to create because they have no dependencies, so they're created first.

Listing 4.2 Linear flow of commands in pseudo language, derived from the dependency graph

```
Create the bucket.                                   Create the database.   Create the virtual machine.
$DB = database create {"password": "***", "engine": "MySQL"}
$VM1 = virtualmachine create {"cpu": 2, "ram": 4, "os": "ubuntu"}
$VM2 = virtualmachine create {"cpu": 2, "ram": 4, "os": "ubuntu"}
$BUCKET = bucket create {}

                                  Wait for the dependencies.        Create the load balancer.
await [$DB, $VM1, $VM2]
$LB = loadbalancer create {"virtualmachines": [$VM1, $VM2]}

await [$LB, $BUCKET]
$CDN = cdn create {...}                  <-- Create the CDN.

await $CDN
$DNS = dns create {...}                  <-- Create the DNS entry.

await $DNS
```

We'll skip the last step—translating the commands from the pseudo language into AWS API calls. You've already learned everything you need to know about infrastructure as code: it's all about dependencies.

Now that you know how important dependencies are to infrastructure as code, let's see how you can use the terminal to create infrastructure. The CLI is one tool for implementing infrastructure as code.

4.2 Using the command-line interface

The AWS CLI is a convenient way to use AWS from your terminal. It runs on Linux, macOS, and Windows and is written in Python. It provides a unified interface for all AWS services. Unless otherwise specified, the output is by default in JSON format.

4.2.1 Why should you automate?

Why should you automate instead of using the graphical AWS Management Console? A script or a blueprint can be reused, and will save you time in the long run. You can build new infrastructures quickly with ready-to-use modules from your former projects, or

automate tasks that you will have to do regularly. Automating your infrastructure creation also enhances the automation of your deployment pipeline.

Another benefit is that a script or blueprint is the most accurate documentation you can imagine (even a computer understands it). If you want to reproduce on Monday what you did last Friday, a script is worth its weight in gold. If you're sick and a coworker needs to take care of your tasks, they will appreciate your blueprints.

You're now going to install and configure the CLI. After that, you can get your hands dirty and start scripting.

4.2.2 Installing the CLI

How you proceed depends on your OS. If you're having difficulty installing the CLI, consult http://mng.bz/N8L6 for a detailed description of many installation options.

Linux and macOS

The CLI requires Python (Python 2.6.5+ or Python 3.3+) and pip. pip is the recommended tool for installing Python packages. To check your Python version, run `python --version` in your terminal. If you don't have Python installed or your version is too old, you'll need to install or update Python before continuing with the next step. To find out if you have already installed pip, run `pip --version` in your terminal. If a version appears, you're fine; otherwise, execute the following to install pip:

```
$ curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py"
$ sudo python get-pip.py
```

Verify your pip installation by running `pip --version` in your terminal again. Now it's time to install the AWS CLI:

```
$ sudo pip install awscli
```

Verify your AWS CLI installation by running `aws --version` in your terminal. The version should be at least `1.11.136`.

Windows

The following steps guide you through installing the AWS CLI on Windows using the MSI Installer:

1. Download the AWS CLI (32-bit or 64-bit) MSI installer from http://aws.amazon.com/cli/.
2. Run the downloaded installer, and install the CLI by going through the installation wizard.
3. Run PowerShell as administrator by searching for "PowerShell" in the Start menu and choosing Run as Administrator from its context menu.
4. Type `Set-ExecutionPolicy Unrestricted` into PowerShell, and press Enter to execute the command. This allows you to execute the unsigned PowerShell scripts from our examples.

5 Close the PowerShell window; you no longer need to work as administrator.
6 Run PowerShell by choosing PowerShell from the Start menu.
7 Verify whether the CLI is working by executing `aws --version` in PowerShell. The version should be at least 1.11.136.

4.2.3 Configuring the CLI

To use the CLI, you need to authenticate. Until now, you've been using the root AWS account. This account can do everything, good and bad. It's strongly recommended that you not use the AWS root account (you'll learn more about security in chapter 6), so let's create a new user.

To create a new user, go to at https://console.aws.amazon.com. Click Services in the navigation bar, and click the IAM (AWS Identity and Access Management) service. A page opens as shown in figure 4.4; select Users at left.

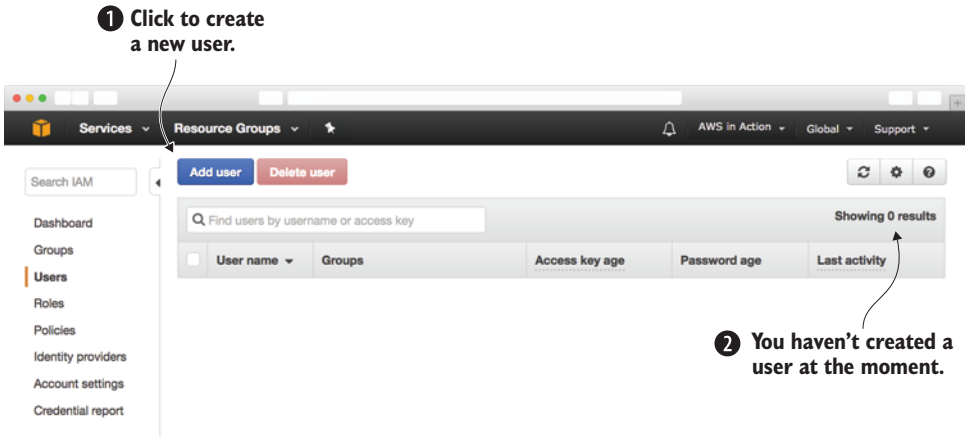


Figure 4.4 IAM users (empty)

Follow these steps to create a user:

1 Click Add User to open the page shown in figure 4.5.
2 Enter `mycli` as the user name.
3 Under Access Type, select Programmatic Access.
4 Click the Next: Permissions button.

In the next step, you have to define the permissions for the new user, as shown in figure 4.6.

1 Click Attach Existing Policies Directly.
2 Select the AdministratorAccess policy.
3 Click the Next: Review button.

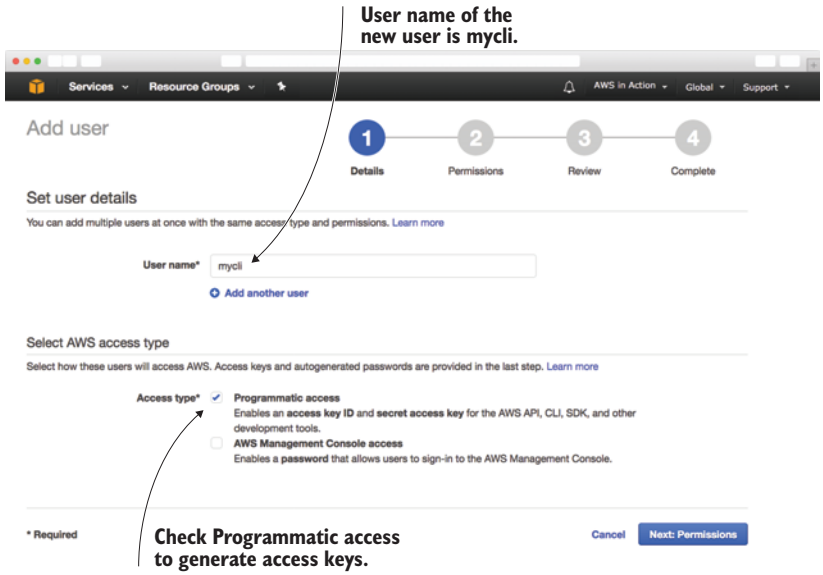


Figure 4.5 Creating an IAM user

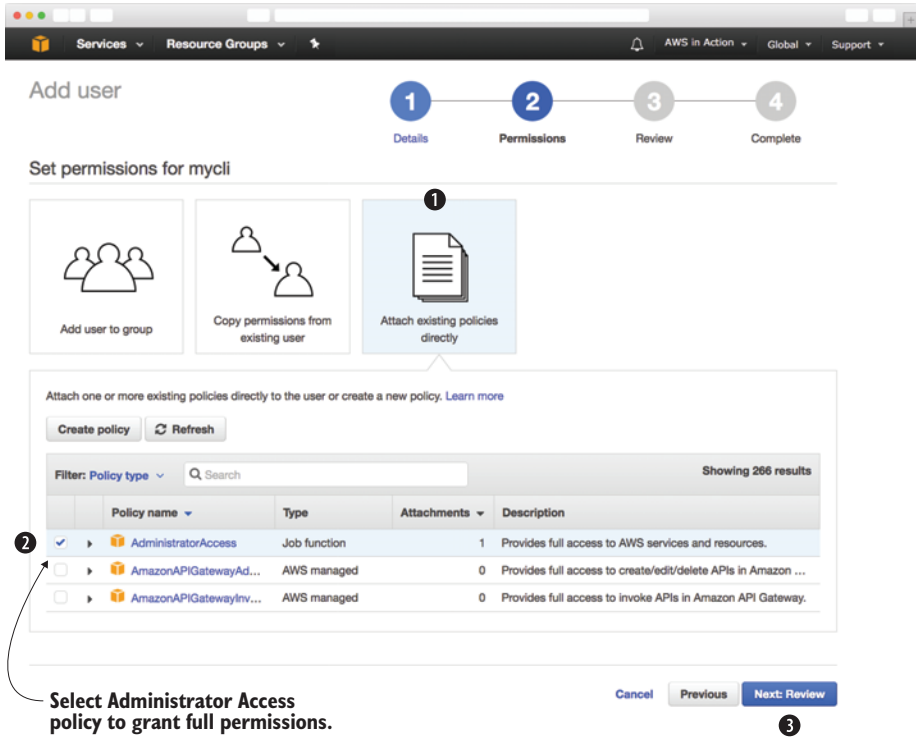


Figure 4.6 Setting permissions for an IAM user

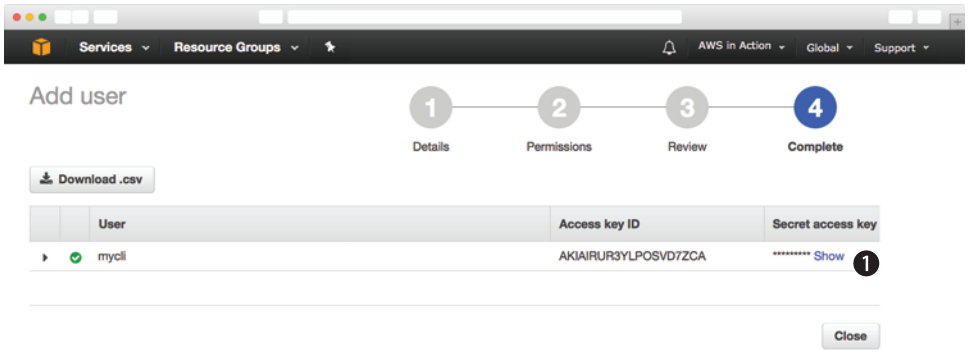


Figure 4.7 Showing access key of an IAM user

The review page sums up what you have configured. Click the Create User button to save. Finally, you will see the page shown in figure 4.7. Click the Show link to display the secret value. You now need to copy the credentials to your CLI configuration. Read on to learn how this works.

Open the terminal on your computer (PowerShell on Windows or a shell on Linux and macOS, not the AWS Management Console), and run `aws configure`. You're asked for four pieces of information:

1. *AWS access key ID*—Copy and paste this value from the Access key ID column (your browser window).
2. *AWS secret access key*—Copy and paste this value from the Secret access key column (your browser window).
3. *Default region name*—Enter `us-east-1`.
4. *Default output format*—Enter `json`.

In the end, the terminal should look similar to this:

Your value will be different! Copy it from your browser window.

```
$ aws configure
AWS Access Key ID [None]:  AKIAIRUR3YLPOSVD7ZCA
AWS Secret Access Key [None]: SSKIng7jkAKERpcT3YphX4cD87sBYgWVw2enqBj7
Default region name [None]: us-east-1
Default output format [None]: json
```

The CLI is now configured to authenticate as the user `mycli`. Switch back to the browser window, and click Close to finish the user-creation wizard.

It's time to test whether the CLI works. Switch to the terminal window, and enter `aws ec2 describe-regions` to get a list of all available regions:

```
$ aws ec2 describe-regions
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-central-1.amazonaws.com",
```

```
      "RegionName": "eu-central-1"
    },
    {
      "Endpoint": "ec2.sa-east-1.amazonaws.com",
      "RegionName": "sa-east-1"
    },
    [...]
    {
      "Endpoint": "ec2.ap-southeast-2.amazonaws.com",
      "RegionName": "ap-southeast-2"
    },
    {
      "Endpoint": "ec2.ap-southeast-1.amazonaws.com",
      "RegionName": "ap-southeast-1"
    }
  ]
}
```

It works! You can now begin to use the CLI.

4.2.4 Using the CLI

Suppose you want to get a list of all running EC2 instances of type t2.micro so you can see what is running in your AWS account. Execute the following command in your terminal:

```
$ aws ec2 describe-instances --filters "Name=instance-type,Values=t2.micro"
{
  "Reservations": []
}
```

Empty list because you haven't created an EC2 instance

To use the AWS CLI, you need to specify a service and an action. In the previous example, the service is `ec2` and the action is `describe-instances`. You can add options with `--key value`:

```
$ aws <service> <action> [--key value ...]
```

One important feature of the CLI is the `help` keyword. You can get help at three levels of detail:

- `aws help`—Shows all available services.
- `aws <service> help`—Shows all actions available for a certain service.
- `aws <service> <action> help`—Shows all options available for the particular service action.

Sometimes you need temporary computing power, like a Linux machine to test something via SSH. To do this, you can write a script that creates a virtual machine for you. The script will run on your local computer and connect to the virtual machine via SSH. After you complete your tests, the script should be able to terminate the virtual machine. The script is used like this:

```
$ ./virtualmachine.sh
waiting for i-c033f117 ...
accepting SSH connections under ec2-54-164-72-62.compute-1.amazonaws.com
ssh -i mykey.pem ec2-user@ec2-54-[...]aws.com
Press [Enter] key to terminate i-c033f117 ...
[...]
terminating i-c033f117 ...
done.
```

Waits until started

SSH connection string

Waits until terminated

Your virtual machine runs until you press the Enter key. When you press Enter, the virtual machine is terminated.

The limitations of this solution are as follows:

- It can handle only one virtual machine at a time.
- There is a different version for Windows than for Linux and macOS.
- It's a command-line application, not graphical.

Nonetheless, the CLI solution solves the following use cases:

- Creating a virtual machine.
- Getting the public name of a virtual machine to connect via SSH.
- Terminating a virtual machine if it's no longer needed.

Depending on your OS, you'll use either Bash (Linux and macOS) or PowerShell (Windows) to script.

One important feature of the CLI needs explanation before you can begin. The `--query` option uses JMESPath, which is a query language for JSON, to extract data from the result. This can be useful because usually you only need a specific field from the result. The following CLI command gets a list of all AMIs in JSON format.

```
$ aws ec2 describe-images
{
  "Images": [
    {
      "ImageId": "ami-146e2a7c",
      "State": "available"
    },
    [...]
    {
      "ImageId": "ami-b66ed3de",
      "State": "available"
    }
  ]
}
```

But to start an EC2 instance, you need the `ImageId` without all the other information. With JMESPath, you can extract just that information. To extract the first `ImageId` property, the path is `Images[0].ImageId`; the result of this query is `"ami-146e2a7c"`.

To extract all `State` properties, the path is `Images[*].State`; the result of this query is `["available", "available"]`.

```
$ aws ec2 describe-images --query "Images[0].ImageId"
"ami-146e2a7c"

$ aws ec2 describe-images --query "Images[0].ImageId" --output text
ami-146e2a7c

$ aws ec2 describe-images --query "Images[*].State"
["available", "available"]
```

With this short introduction to JMESPath, you're well equipped to extract the data you need.

> **Where is the code located?**
> All code can be found in the book's code repository on GitHub: https://github.com/AWSinAction/code2. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip.

Linux and macOS can interpret Bash scripts, whereas Windows prefers PowerShell scripts. So, we've created two versions of the same script.

Linux and macOS

You can find the following listing in `/chapter04/virtualmachine.sh` in the book's code folder. You can run it either by copying and pasting each line into your terminal or by executing the entire script via `chmod +x virtualmachine.sh && ./virtualmachine.sh`.

Listing 4.3 Creating and terminating a virtual machine from the CLI (Bash)

```
#!/bin/bash -e
AMIID="$(aws ec2 describe-images --filters \
➥ "Name=name,Values=amzn-ami-hvm-2017.09.1.*-x86_64-gp2" \
➥ --query "Images[0].ImageId" --output text)"
VPCID="$(aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
➥ --query "Vpcs[0].VpcId" --output text)"
SUBNETID="$(aws ec2 describe-subnets --filters "Name=vpc-id, Values=$VPCID" \
➥ --query "Subnets[0].SubnetId" --output text)"
SGID="$(aws ec2 create-security-group --group-name mysecuritygroup \
➥ --description "My security group" --vpc-id "$VPCID" --output text)"
aws ec2 authorize-security-group-ingress --group-id "$SGID" \
➥ --protocol tcp --port 22 --cidr 0.0.0.0/0
INSTANCEID="$(aws ec2 run-instances --image-id "$AMIID" --key-name mykey \
```

-e makes Bash abort if a command fails.

Get the ID of Amazon Linux AMI.

Get the default VPC ID.

Get the default subnet ID.

Create a Security Group.

Allow inbound SSH connections.

Create and start the virtual machine.

```
    --instance-type t2.micro --security-group-ids "$SGID" \
    --subnet-id "$SUBNETID" --query "Instances[0].InstanceId" --output text)"
echo "waiting for $INSTANCEID ..."
aws ec2 wait instance-running --instance-ids "$INSTANCEID"
PUBLICNAME="$(aws ec2 describe-instances --instance-ids "$INSTANCEID" \
    --query "Reservations[0].Instances[0].PublicDnsName" --output text)"
echo "$INSTANCEID is accepting SSH connections under $PUBLICNAME"
echo "ssh -i mykey.pem ec2-user@$PUBLICNAME"
read -r -p "Press [Enter] key to terminate $INSTANCEID ..."
aws ec2 terminate-instances --instance-ids "$INSTANCEID"
echo "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids "$INSTANCEID"
aws ec2 delete-security-group --group-id "$SGID"
```

Wait until the virtual machine is started.

Get the public name of virtual machine.

Terminate the virtual machine.

Wait until the virtual machine is terminated.

Delete the Security Group.

Cleaning up

Make sure you terminate the virtual machine before you go on!

WINDOWS

You can find the following listing in `/chapter04/virtualmachine.ps1` in the book's code folder. Right-click the virtualmachine.ps1 file, and select Run with PowerShell to execute the script.

Listing 4.4 Creating and terminating a virtual machine from the CLI (PowerShell)

```
$ErrorActionPreference = "Stop"
$AMIID=aws ec2 describe-images --filters \
    "Name=name,Values=amzn-ami-hvm-2017.09.1.*-x86_64-gp2" \
    --query "Images[0].ImageId" --output text
$VPCID=aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
    --query "Vpcs[0].VpcId" --output text
$SUBNETID=aws ec2 describe-subnets --filters "Name=vpc-id, Values=$VPCID" \
    --query "Subnets[0].SubnetId" --output text
$SGID=aws ec2 create-security-group --group-name mysecuritygroup \
    --description "My security group" --vpc-id $VPCID \
    --output text
aws ec2 authorize-security-group-ingress --group-id $SGID \
    --protocol tcp --port 22 --cidr 0.0.0.0/0
$INSTANCEID=aws ec2 run-instances --image-id $AMIID --key-name mykey \
    --instance-type t2.micro --security-group-ids $SGID \
    --subnet-id $SUBNETID \
    --query "Instances[0].InstanceId" --output text
```

Get the default subnet ID.

Abort if the command fails.

Get the ID of Amazon Linux AMI.

Get the default VPC ID.

Create the Security Group.

Allow inbound SSH connections.

Create and start the virtual machine.

```
Write-Host "waiting for $INSTANCEID ..."
aws ec2 wait instance-running --instance-ids $INSTANCEID
$PUBLICNAME=aws ec2 describe-instances --instance-ids $INSTANCEID \
➥ --query "Reservations[0].Instances[0].PublicDnsName" --output text
Write-Host "$INSTANCEID is accepting SSH under $PUBLICNAME"
Write-Host "connect to $PUBLICNAME via SSH as user ec2-user"
Write-Host "Press [Enter] key to terminate $INSTANCEID ..."
Read-Host
aws ec2 terminate-instances --instance-ids $INSTANCEID
Write-Host "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids $INSTANCEID
aws ec2 delete-security-group --group-id $SGID
```

Wait until the virtual machine is started.

Get the public name of virtual machine.

Terminate the virtual machine.

Wait until the virtual machine is terminated

Delete the Security Group.

Cleaning up

Make sure you terminate the virtual machine before you go on!

4.3 Programming with the SDK

AWS offers SDKs for a number of programming languages and platforms:

- Android
- Browsers (JavaScript)
- iOS
- Java
- .NET
- Node.js (JavaScript)
- PHP
- Python
- Ruby
- Go
- C++

An AWS SDK is a convenient way to make calls to the AWS API from your favorite programming language. The SDK takes care of things like authentication, retry on error, HTTPS communication, and XML or JSON (de)serialization. You're free to choose the SDK for your favorite language, but in this book most examples are written in JavaScript and run in the Node.js runtime environment.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.

After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should response with something similar to `v8.*`. Now you're ready to run JavaScript examples, like the Node Control Center for AWS.

(continued)

Your Node.js installation comes with a important tool called *npm*, which is the package manager for Node.js. Verify the installation by running `npm --version` in your terminal.

To run a JavaScript script in Node.js, enter `node` *script.js* in your terminal, where *script.js* is the name of the script file. We use Node.js in this book because it's easy to install, it requires no IDE, and the syntax is familiar to most programmers.

Don't be confused by the terms JavaScript and Node.js. If you want to be precise, JavaScript is the language and Node.js is the runtime environment. But don't expect anybody to make that distinction. Node.js is also called node.

Do you want to get started with Node.js? We recommend *Node.js in Action (2nd Edition)* from Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* from PJ Evans, (Manning, 2018).

To understand how the AWS SDK for Node.js (JavaScript) works, let's create a Node.js (JavaScript) application that controls EC2 instances via the AWS SDK.

4.3.1 Controlling virtual machines with SDK: nodecc

The *Node Control Center for AWS (nodecc)* is for managing multiple temporary EC2 instances using a text UI written in JavaScript. nodecc has the following features:

- It can handle multiple virtual machines.
- It's written in JavaScript and runs in Node.js, so it's portable across platforms.
- It uses a textual UI.

Figure 4.8 shows what nodecc looks like. You can find the nodecc application at `/chapter04/nodecc/` in the book's code folder. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies. To start nodecc, run

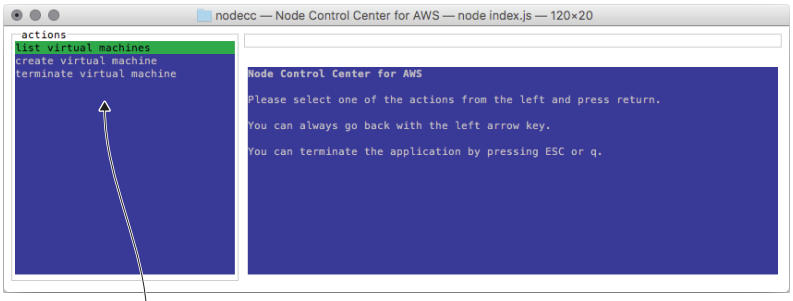


Figure 4.8 nodecc: start screen

`node index.js`. You can always go back by pressing the left arrow key. You can quit the application by pressing Esc or q. The SDK uses the same settings you created for the CLI, so you're using the `mycli` user when running nodecc.

4.3.2 How nodecc creates a virtual machine

Before you can do anything with nodecc, you need at least one virtual machine. To start a virtual machine, choose the AMI you want, as figure 4.9 shows.

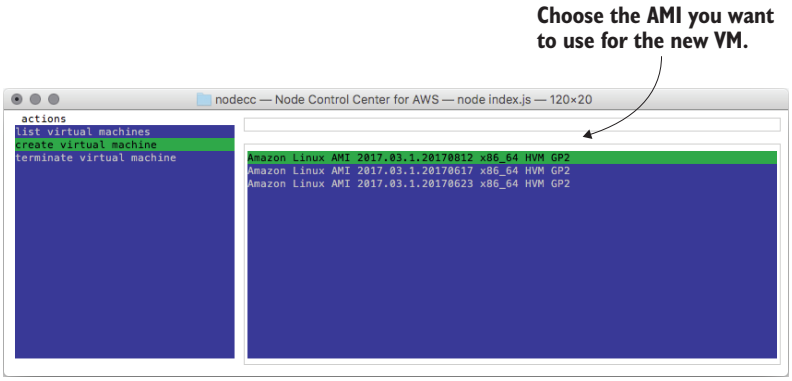


Figure 4.9 nodecc: creating a virtual machine (step 1 of 2)

The code that fetches the list of the available AMIs is located at `lib/listAMIs.js`.

Listing 4.5 Fetching the list of available AMIs /lib/listAMIs.js

```
const jmespath = require('jmespath');
const AWS = require('aws-sdk');
const ec2 = new AWS.EC2({region: 'us-east-1'});

module.exports = (cb) => {
  ec2.describeImages({
    Filters: [{
      Name: 'name',
      Values: ['amzn-ami-hvm-2017.09.1.*-x86_64-gp2']
    }]
  }, (err, data) => {
    if (err) {
      cb(err);
    } else {
      const amiIds = jmespath.search(data, 'Images[*].ImageId');
      const descriptions = jmespath.search(data, 'Images[*].Description');
      cb(null, {amiIds: amiIds, descriptions: descriptions});
    }
  });
};
```

Configure an EC2 endpoint.

Require is used to load modules.

module.exports makes this function available to users of the listAMIs module.

Action

In case of failure, err is set.

Otherwise, data contains all AMIs.

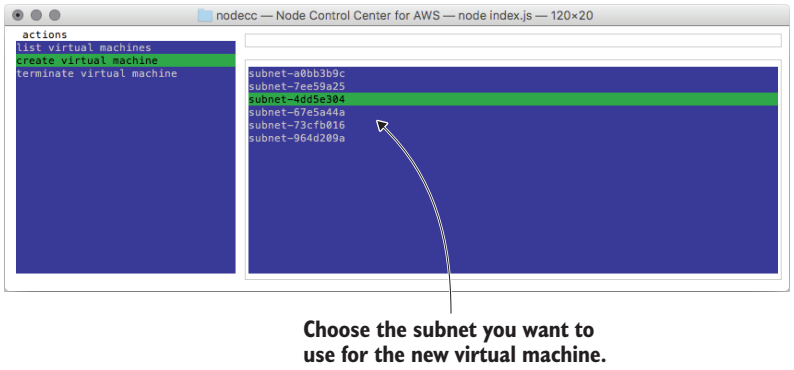


Figure 4.10 nodecc: creating a virtual machine (step 2 of 2)

The code is structured in such a way that each action is implemented in the lib folder. The next step to create a virtual machine is to choose which subnet the virtual machine should be started in. You haven't learned about subnets yet, so for now select one randomly; see figure 4.10. The corresponding script is located at `lib/listSubnets.js`.

After you select the subnet, the virtual machine is created by `lib/createVM.js`, and you see a starting screen. Now it's time to find out the public name of the newly created virtual machine. Use the left arrow key to switch to the navigation section.

4.3.3 How nodecc lists virtual machines and shows virtual machine details

One important use case that nodecc must support is showing the public name of a VM that you can use to connect via SSH. Because nodecc handles multiple virtual machines, the first step is to select a VM, as shown in figure 4.11.

Look at `lib/listVMs.js` to see how a list of virtual machines can be retrieved with the AWS SDK. After you select the VM, you can display its details; see figure 4.12. You could use the `PublicDnsName` to connect to the EC2 instance via SSH. Press the left arrow key to switch back to the navigation section.

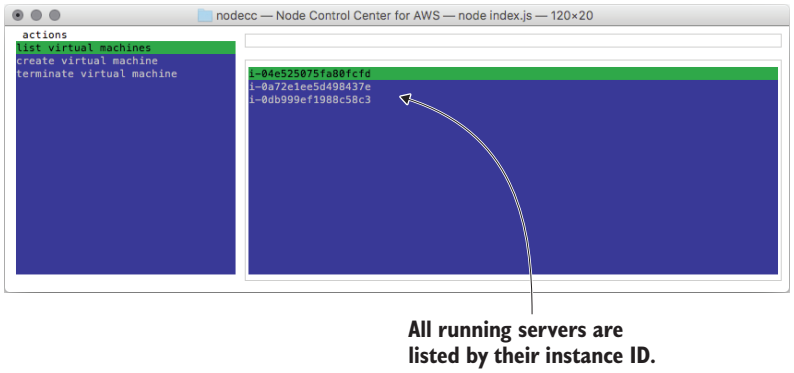


Figure 4.11 nodecc: listing virtual machines

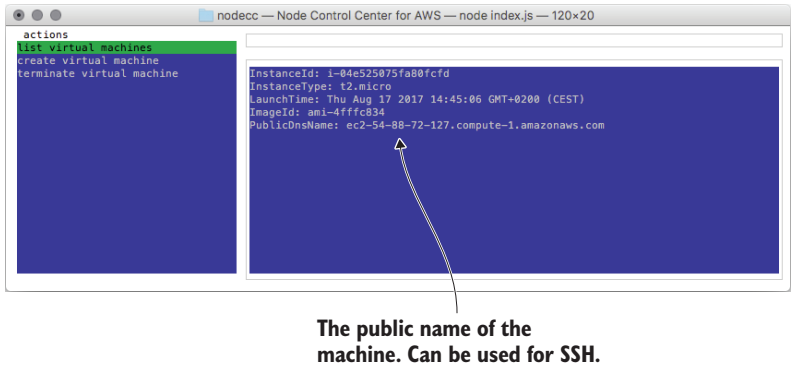


Figure 4.12 nodecc: showing virtual machine details

4.3.4 How nodecc terminates a virtual machine

To terminate a virtual machine, you first have to select it. To list the virtual machines, use `lib/listVMs.js` again. After the VM is selected, `lib/terminateVM.js` takes care of termination.

That's nodecc: a text UI program for controlling temporary EC2 instances. Take some time to think about what you could create by using your favorite language and the AWS SDK. Chances are high that you might come up with a new business idea!

Cleaning up

Make sure you terminate all started virtual machines before you go on!

4.4 Using a blueprint to start a virtual machine

Earlier, we talked about JIML to introduce the concept of infrastructure as code. Luckily, AWS already offers a tool that does much better than JIML: *AWS CloudFormation*. CloudFormation is based on templates, which up to now we've called blueprints.

NOTE We use the term *blueprint* when discussing infrastructure automation in general. Blueprints used for AWS CloudFormation, a configuration management service, are called *templates*.

A *template* is a description of your infrastructure, written in JSON or YAML, that can be interpreted by CloudFormation. The idea of describing something rather than listing the necessary actions is called a *declarative approach*. Declarative means you tell CloudFormation how your infrastructure should look. You aren't telling CloudFormation what actions are needed to create that infrastructure, and you don't specify the sequence in which the actions need to be executed.

The benefits of CloudFormation are as follows:

- *It's a consistent way to describe infrastructure on AWS.* If you use scripts to create your infrastructure, everyone will solve the same problem differently. This is a hurdle for new developers and operators trying to understand what the code is doing. CloudFormation templates are a clear language for defining infrastructure.
- *It can handle dependencies.* Ever tried to register a web server with a load balancer that wasn't yet available? At first glance, you'll miss a lot of dependencies. Trust us: never try to set up complex infrastructure using scripts. You'll end up in dependency hell!
- *It's replicable.* Is your test environment an exact copy of your production environment? Using CloudFormation, you can create two identical infrastructures and keep them in sync.
- *It's customizable.* You can insert custom parameters into CloudFormation to customize your templates as you wish.
- *It's testable.* If you can create your architecture from a template, it's testable. Just start a new infrastructure, run your tests, and shut it down again.
- *It's updatable.* CloudFormation supports updates to your infrastructure. It will figure out the parts of the template that have changed and apply those changes as smoothly as possible to your infrastructure.
- *It minimizes human failure.* CloudFormation doesn't get tired—even at 3 a.m.
- *It's the documentation for your infrastructure.* A CloudFormation template is a JSON or YAML document. You can treat it as code and use a version control system like Git to keep track of the changes.
- *It's free.* Using CloudFormation comes at no additional charge. If you subscribe to an AWS support plan, you also get support for CloudFormation.

We think CloudFormation is one of the most powerful tools available for managing infrastructure on AWS.

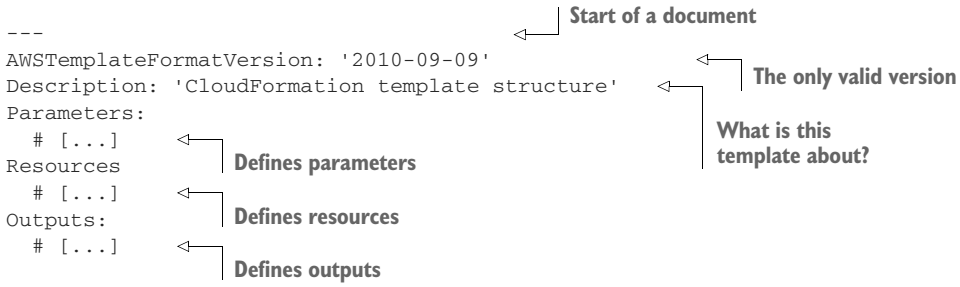
4.4.1 Anatomy of a CloudFormation template

A basic CloudFormation template is structured into five parts:

1. *Format version*—The latest template format version is 2010-09-09, and this is currently the only valid value. Specify this version; the default is to use the latest version, which will cause problems if new versions are introduced in the future.
2. *Description*—What is this template about?
3. *Parameters*—Parameters are used to customize a template with values: for example, domain name, customer ID, and database password.
4. *Resources*—A resource is the smallest block you can describe. Examples are a virtual machine, a load balancer, or an Elastic IP address.
5. *Outputs*—An output is comparable to a parameter, but the other way around. An output returns something from your template, such as the public name of an EC2 instance.

A basic template looks like the following listing.

Listing 4.6 CloudFormation template structure



Let's take a closer look at parameters, resources, and outputs.

Format version and description

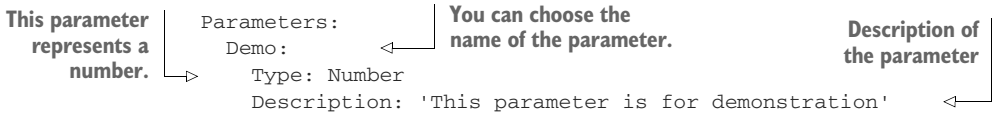
The only valid `AWSTemplateFormatVersion` value at the moment is `2010-09-09`. Always specify the format version. If you don't, CloudFormation will use whatever version is the latest one. As mentioned earlier, this means that if a new format version is introduced in the future, you'll end up using the wrong format and get into serious trouble.

`Description` isn't mandatory, but we encourage you to take some time to document what the template is about. A meaningful description will help you in the future to remember what the template is for. It will also help your coworkers.

Parameters

A parameter has at least a name and a type. We encourage you to add a description as well.

Listing 4.7 CloudFormation parameter structure



Valid types are listed in table 4.1.

Table 4.1 CloudFormation parameter types

Type	Description
`String` `CommaDelimitedList`	A string or a list of strings separated by commas
`Number` `List<Number>`	An integer or float, or a list of integers or floats

Table 4.1 CloudFormation parameter types *(continued)*

Type	Description
`AWS::EC2::AvailabilityZone::Name` `List<AWS::EC2::AvailabilityZone::Name>`	An Availability Zone, such as us-west-2a, or a list of Availability Zones
`AWS::EC2::Image::Id` `List<AWS::EC2::Image::Id>`	An AMI ID or a list of AMIs
`AWS::EC2::Instance::Id` `List<AWS::EC2::Instance::Id>`	An EC2 instance ID or a list of EC2 instance IDs
`AWS::EC2::KeyPair::KeyName`	An Amazon EC2 key-pair name
`AWS::EC2::SecurityGroup::Id` `List<AWS::EC2::SecurityGroup::Id>`	A security group ID or a list of security group IDs
`AWS::EC2::Subnet::Id` `List<AWS::EC2::Subnet::Id>`	A subnet ID or a list of subnet IDs
`AWS::EC2::Volume::Id` `List<AWS::EC2::Volume::Id>`	An EBS volume ID (network attached storage) or a list of EBS volume IDs
`AWS::EC2::VPC::Id` `List<AWS::EC2::VPC::Id>`	A VPC ID (virtual private cloud) or a list of VPC IDs
`AWS::Route53::HostedZone::Id` `List<AWS::Route53::HostedZone::Id>`	A DNS zone ID or a list of DNS zone IDs

In addition to using the `Type` and `Description` properties, you can enhance a parameter with the properties listed in table 4.2.

Table 4.2 CloudFormation parameter properties

Property	Description	Example
`Default`	A default value for the parameter	`Default: 'm5.large'`
`NoEcho`	Hides the parameter value in all graphical tools (useful for passwords)	`NoEcho: true`
`AllowedValues`	Specifies possible values for the parameter	`AllowedValues: [1, 2, 3]`
`AllowedPattern`	More generic than `AllowedValues` because it uses a regular expression	`AllowedPattern: '[a-zA-Z0-9]*'` allows only a–z, A–Z, and 0–9 with any length
`MinLength`, `MaxLength`	Defines how long a parameter can be	`MinLength: 12`
`MinValue`, `MaxValue`	Used in combination with the `Number` type to define lower and upper bounds	`MaxValue: 10`
`ConstraintDescription`	A string that explains the constraint when the constraint is violated.	`ConstraintDescription: 'Maximum value is 10.'`

A parameter section of a CloudFormation template could look like this:

```
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'    <-- Only key pair names are allowed.
  NumberOfVirtualMachines:
    Description: 'How many virtual machine do you like?'
    Type: Number
    Default: 1                <-- The default is one virtual machine.
    MinValue: 1
    MaxValue: 5           <-- Prevent massive costs with an upper bound.
  WordPressVersion:
    Description: 'Which version of WordPress do you want?'
    Type: String
    AllowedValues: ['4.1.1', '4.0.1']      <-- Restricted to certain versions
```

Now you should have a better feel for parameters. If you want to know everything about them, see the documentation at http://mng.bz/jg7B or follow along in the book and learn by doing.

Resources

A resource has at least a name, a type, and some properties.

Listing 4.8 CloudFormation resources structure

```
Resources:
  VM:                                  <-- Name or logical ID of the resource that you can choose
    Type: 'AWS::EC2::Instance'         <-- Defines an EC2 instance
    Properties:
      # [...]                          <-- Properties needed for the type of resource
```

When defining resources, you need to know about the type and that type's properties. In this book, you'll get to know a lot of resource types and their respective properties. An example of a single EC2 instance follows. If you see `!Ref` *NameOfSomething*, think of it as a placeholder for what is referenced by the name. You can reference parameters and resources to create dependencies.

Listing 4.9 CloudFormation EC2 instance resource

```
Resources:
  VM:                                  <-- Name or logical ID of the resource that you can choose
    Type: 'AWS::EC2::Instance'         <-- Defines an EC2 instance
    Properties:
      ImageId: 'ami-6057e21a'          <-- Some hard-coded settings
      InstanceType: 't2.micro'
      KeyName: mykey
      NetworkInterfaces:
       - AssociatePublicIpAddress: true
```

```
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - 'sg-123456'
        SubnetId: 'subnet-123456'
```

Now you've described the virtual machine, but how can you output its public name?

OUTPUTS

A CloudFormation template's output includes at least a name (like parameters and resources) and a value, but we encourage you to add a description as well. You can use outputs to pass data from within your template to the outside.

Listing 4.10 CloudFormation outputs structure

```
Outputs:
  NameOfOutput:
    Value: '1'
    Description: 'This output is always 1'
```

Value of the output

Name of the output that you can choose

Static outputs like this one aren't very useful. You'll mostly use values that reference the name of a resource or an attribute of a resource, like its public name.

Listing 4.11 CloudFormation outputs example

```
Outputs:
  ID:
    Value: !Ref Server
    Description: 'ID of the EC2 instance'
  PublicName:
    Value: !GetAtt 'Server.PublicDnsName'
    Description: 'Public name of the EC2 instance'
```

References the EC2 instance

Get the attribute PublicDnsName of the EC2 instance.

You'll get to know the most important attributes of `!GetAtt` later in the book. If you want to know about all of them, see http://mng.bz/q5I4.

Now that we've taken a brief look at the core parts of a CloudFormation template, it's time to make one of your own.

4.4.2 Creating your first template

How do you create a CloudFormation template? Different options are available:

- Use a text editor or IDE to write a template from scratch.
- Use the CloudFormation Designer, a graphical user interface offered by AWS.
- Start with a template from a public library that offers a default implementation and adapt it to your needs.
- Use CloudFormer, a tool for creating a template based on an existing infrastructure provided by AWS.
- Use a template provided by your vendor.

AWS and their partners offer CloudFormation templates for deploying popular solutions: AWS Quick Starts at https://aws.amazon.com/quickstart/. Furthermore, we have open sourced the templates we are using in our day-to-day work on GitHub: https://github.com/widdix/aws-cf-templates.

Suppose you've been asked to provide a VM for a developer team. After a few months, the team realizes the VM needs more CPU power, because the usage pattern has changed. You can handle that request with the CLI and the SDK, but as you learned in section 3.4, before the instance type can be changed, you must stop the EC2 instance. The process will be as follows:

1. Stop the instance.
2. Wait for the instance to stop.
3. Change the instance type.
4. Start the instance.
5. Wait for the instance to start.

A declarative approach like that used by CloudFormation is simpler: just change the `InstanceType` property and update the template. `InstanceType` can be passed to the template via a parameter. That's it! You can begin creating the template.

Listing 4.12 Template to create an EC2 instance with CloudFormation

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 4'
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
  VPC:
    # [...]
  Subnet:
    # [...]
  InstanceType:
    Description: 'Select one of the possible instance types'
    Type: String
    Default: 't2.micro'
    AllowedValues": ['t2.micro', 't2.small', 't2.medium']
Resources:
  SecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      # [...]
  VM:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyName
      NetworkInterfaces:
```

The user defines which key to use.

You'll learn about this in section 6.5.

You'll learn about this in section 6.5.

The user defines the instance type.

You'll learn about this in section 6.4.

Defines a minimal EC2 instance

```
        - AssociatePublicIpAddress: true
          DeleteOnTermination: true
          DeviceIndex: 0
          GroupSet:
          - !Ref SecurityGroup
          SubnetId: !Ref Subnet
Outputs:                                    Returns the public name
  PublicName:                               of the EC2 instance
    Value: !GetAtt 'Server.PublicDnsName'
    Description: 'Public name (connect via SSH as user ec2-user)'
```

You can find the full code for the template at `/chapter04/virtualmachine.yaml` in the book's code folder. Please don't worry about VPC, subnets, and security groups at the moment; you'll get to know them in chapter 6.

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at `chapter04/virtualmachine.yaml`. On S3, the same file is located at http://mng.bz/B5UM.

If you create an infrastructure from a template, CloudFormation calls it a *stack*. You can think of *template* versus *stack* much like *class* versus *object*. The template exists only once, whereas many stacks can be created from the same template.

Open the AWS Management Console at https://console.aws.amazon.com. Click Services in the navigation bar, and then click the CloudFormation service. Figure 4.13 shows the initial CloudFormation screen with an overview of all the stacks.

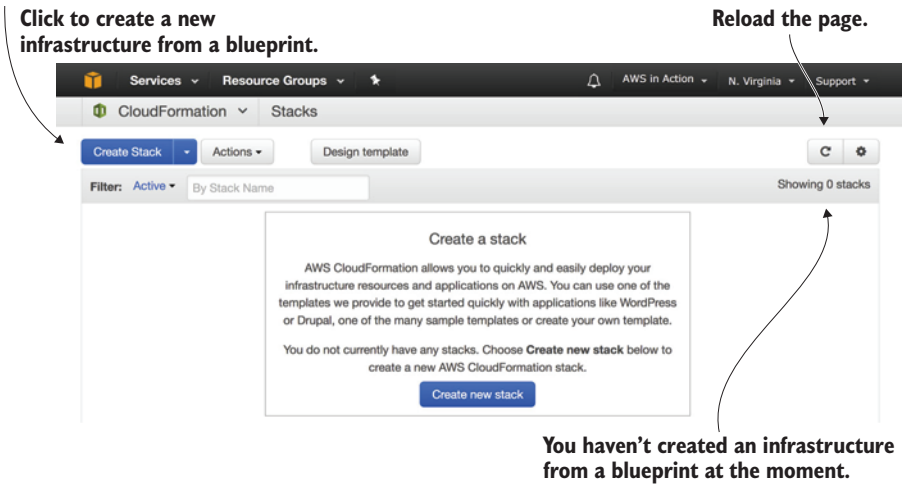


Figure 4.13 Overview of CloudFormation stacks

The following steps will guide you through creating your stack:

1. Click Create Stack to start a four-step wizard.
2. Select Specify an Amazon S3 Template URL, and enter `https://s3.amazonaws.com/awsinaction-code2/chapter04/virtualmachine.yaml` as shown in figure 4.14.

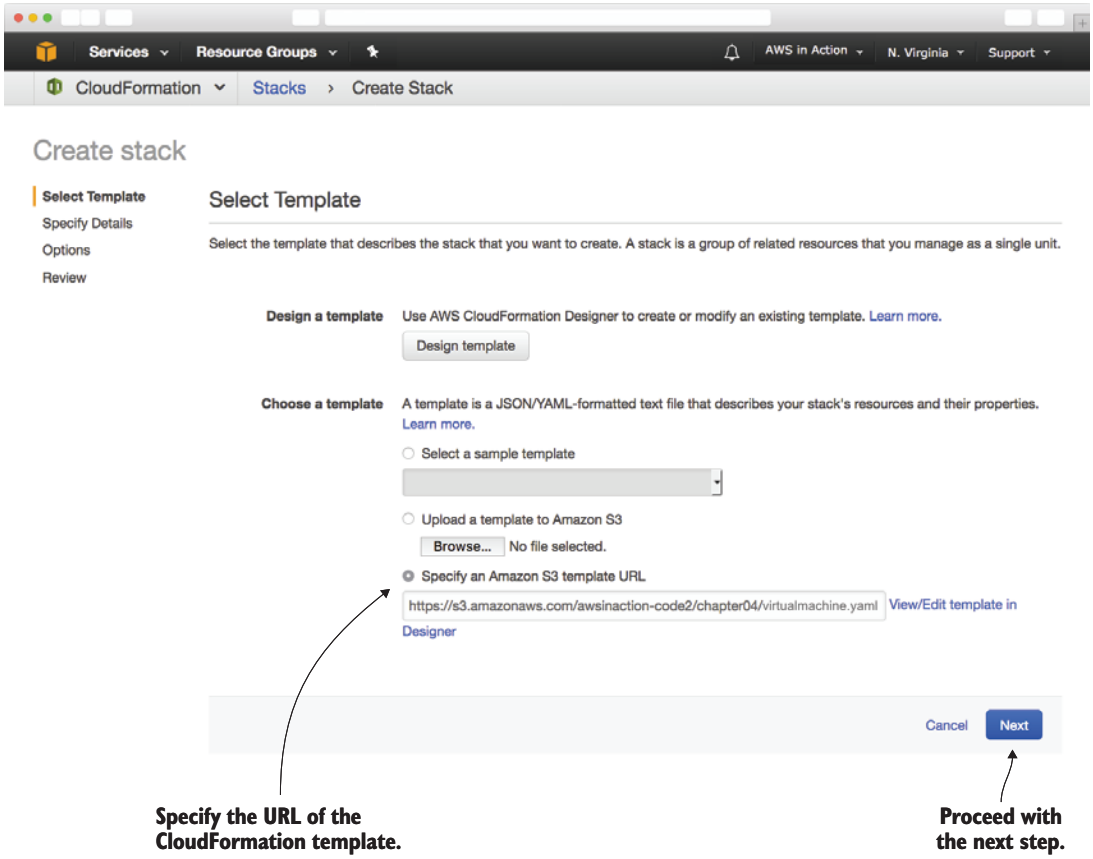


Figure 4.14 Creating a CloudFormation stack: selecting a template (step 1 of 4)

In the second step, you define the stack name and parameters. Give the stack a name like `server` and fill out the parameter values:

1. `InstanceType`—Select t2.micro.
2. `KeyName`—Select mykey.
3. `Subnet`—Select the first value in the drop-down list. You'll learn about subnets later.
4. `VPC`—Select the first value in the drop-down list. You'll learn about VPCs later.

Figure 4.15 shows the parameters step. Click Next after you've chosen a value for every parameter, to proceed with the next step.

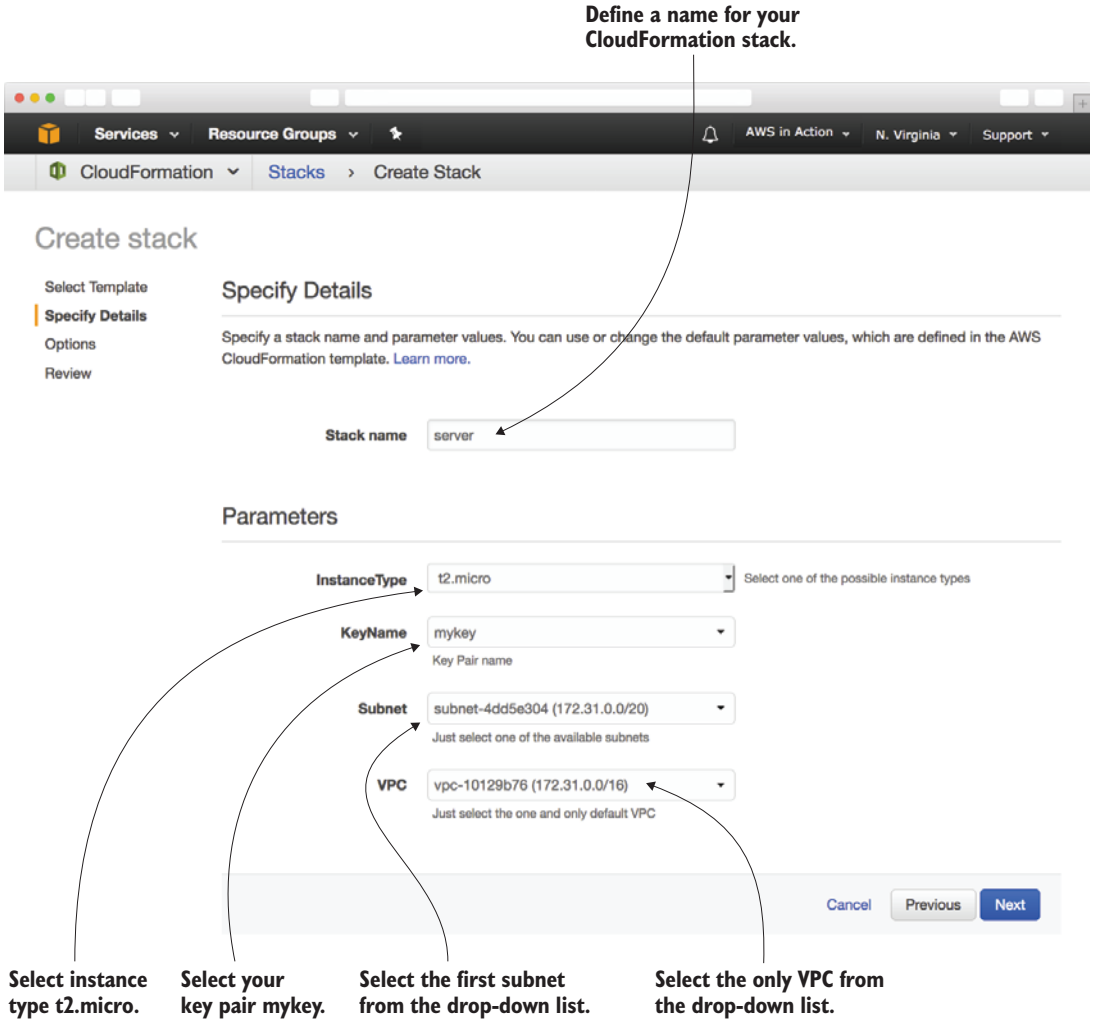


Figure 4.15 Creating a CloudFormation stack: defining parameters (step 2 of 4)

In the third step, you can define optional tags for the stack and advanced configuration. You can skip this step at this point in the book, because you will not use any advances features for now. All resources created by the stack will be tagged by CloudFormation by default. Click Next to go to the last step.

Step four displays a summary of the stack, as shown in figure 4.16.

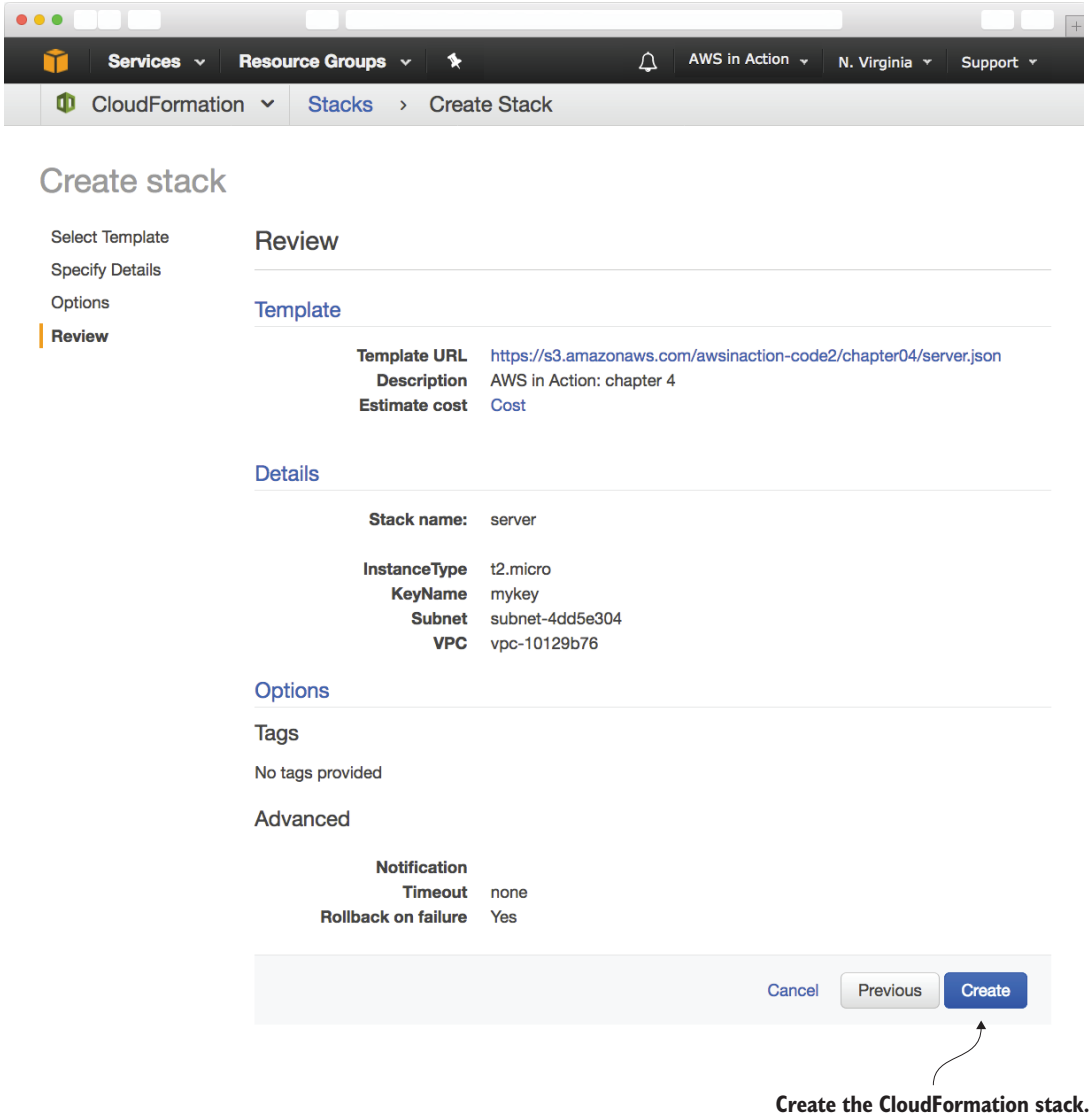


Figure 4.16 Creating a CloudFormation stack: summary (step 4 of 4)

Click Create. CloudFormation now starts to create the stack. If the process is successful, you'll see the screen shown in figure 4.17. As long as Status is `CREATE_IN_PROGRESS`, you need to be patient. When Status is `CREATE_COMPLETE`, select the stack and click the Outputs tab to see the public name of the EC2 instance.

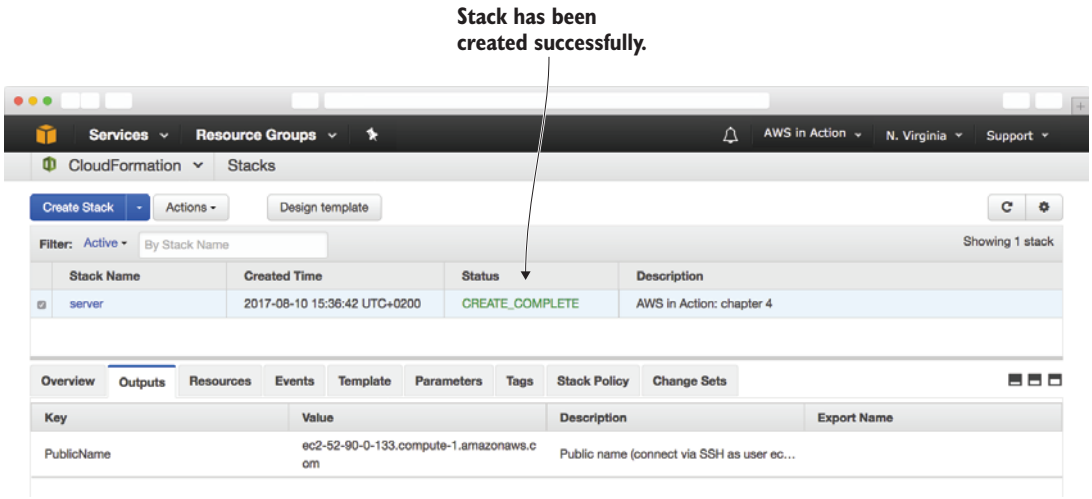


Figure 4.17 The CloudFormation stack has been created.

It's time to test modifying the instance type. Select the stack, and click the Update Stack button. The wizard that starts is similar to the one you used during stack creation. Figure 4.18 shows the first step of the wizard. Select Use Current Template and proceed with the next step.

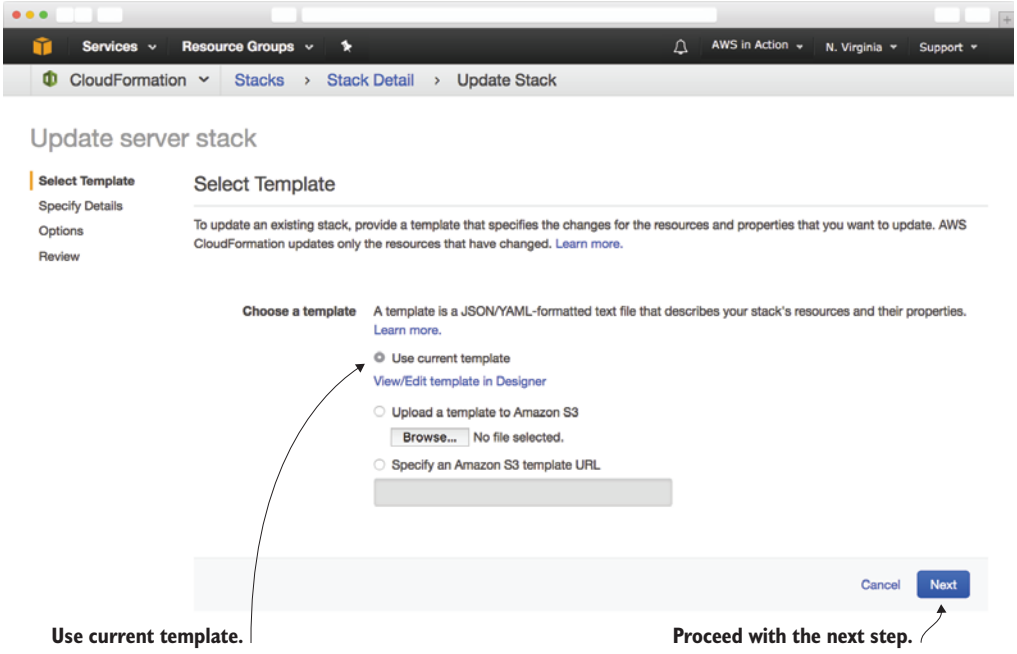


Figure 4.18 Updating the CloudFormation stack: summary (step 1 of 4)

In step 2, you need to change the InstanceType parameter value: choose t2.small to double or t2.medium to quadruple the computing power of your EC2 instance.

> **WARNING** Starting a virtual machine with instance type t2.small or t2.medium will incur charges. See http://aws.amazon.com/ec2/pricing to find out the current hourly price.

Step 3 is about sophisticated options during the update of the stack. You don't need any of these features now, so skip the step by clicking Next. Step 4 is a summary; click Update. The stack now has Status `UPDATE_IN_PROGRESS`. After a few minutes, Status should change to `UPDATE_COMPLETE`. You can select the stack and get the public name of a new EC2 instance with the increased instance type by looking at the Outputs tab.

Alternatives to CloudFormation

If you don't want to write plain JSON or YAML to create templates for your infrastructure, there are a few alternatives to CloudFormation. Tools like Troposphere, a library written in Python, help you to create CloudFormation templates without having to write JSON or YAML. They add another abstraction level on top of CloudFormation to do so.

There are also tools that allow you to use infrastructure as code without needing CloudFormation. Terraform (https://www.terraform.io/) let you describe your infrastructure as code, for example.

When you changed the parameter, CloudFormation figured out what needed to be done to achieve the end result. That's the power of a declarative approach: you say what the end result should look like, not how the end result should be achieved.

Cleaning up

Delete the stack by selecting it and clicking the Delete Stack button.

Summary

- Use the CLI, one of the SDKs, or CloudFormation to automate your infrastructure on AWS.
- Infrastructure as code describes the approach of programming the creation and modification of your infrastructure, including virtual machines, networking, storage, and more.
- You can use the CLI to automate complex processes in AWS with scripts (Bash and PowerShell).

- You can use SDKs for nine programming languages and platforms to embed AWS into your applications and create applications like nodecc.
- CloudFormation uses a declarative approach in JSON or YAML: you only define the end state of your infrastructure, and CloudFormation figures out how this state can be achieved. The major parts of a CloudFormation template are parameters, resources, and outputs.

5 Automating deployment: CloudFormation, Elastic Beanstalk, and OpsWorks

This chapter covers

- Creating VMs and running scripts on startup with AWS CloudFormation
- Deploying common web apps with AWS Elastic Beanstalk
- Deploying multilayer apps with AWS OpsWorks
- Comparing the different deployment services on AWS

Whether you want to use software from in-house development, open source projects, or commercial vendors, you need to install, update, and configure the application and its dependencies. This process is called *deployment*. In this chapter, you'll learn about three tools for deploying applications to virtual machines on AWS:

1. Deploying a VPN solution with the help of AWS CloudFormation and a script that starts at the end of the boot process.

2. Deploying a collaborative text editor with AWS Elastic Beanstalk. The text editor *Etherpad* is a simple web application and a perfect fit for AWS Elastic Beanstalk, because it supports Node.js by default.
3. Deploying an IRC web client and IRC server with AWS OpsWorks. The setup consists of two parts: the IRC web client and the IRC server itself. Our example consists of multiple layers and is perfect for AWS OpsWorks.

We've chosen examples that don't need a storage solution for this chapter, but all three deployment solutions would support delivering an application together with a storage solution. You'll find examples using storage in part 3 of the book.

> **Examples are 100% covered by the Free Tier**
>
> The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

What steps are required to deploy a typical web application like WordPress—a widely used blogging platform—to a virtual machine?

1. Install an Apache HTTP server, a MySQL database, a PHP runtime environment, a MySQL library for PHP, and an SMTP mail server.
2. Download the WordPress application, and unpack the archive on your server.
3. Configure the Apache web server to serve the PHP application.
4. Configure the PHP runtime environment to tweak performance and increase security.
5. Edit the wp-config.php file to configure the WordPress application.
6. Edit the configuration of the SMTP server, and make sure mail can only be sent from the virtual machine, to avoid misuse from spammers.
7. Start the MySQL, SMTP, and HTTP services.

Steps 1–2 handle installing and updating the executables. These executables are configured in steps 3–6. Step 7 starts the services.

System administrators working with a traditional infrastructure often perform these steps manually by following how-to guides. Deploying applications manually is no longer recommended in a flexible cloud environment. Instead your goal will be to automate these steps with the help of the tools you'll discover next.

5.1 Deploying applications in a flexible cloud environment

If you want to take advantage of cloud features like scaling the number of machines depending on the current load or building a highly available infrastructure, you'll need to start new virtual machines several times a day. On top of that, the number of

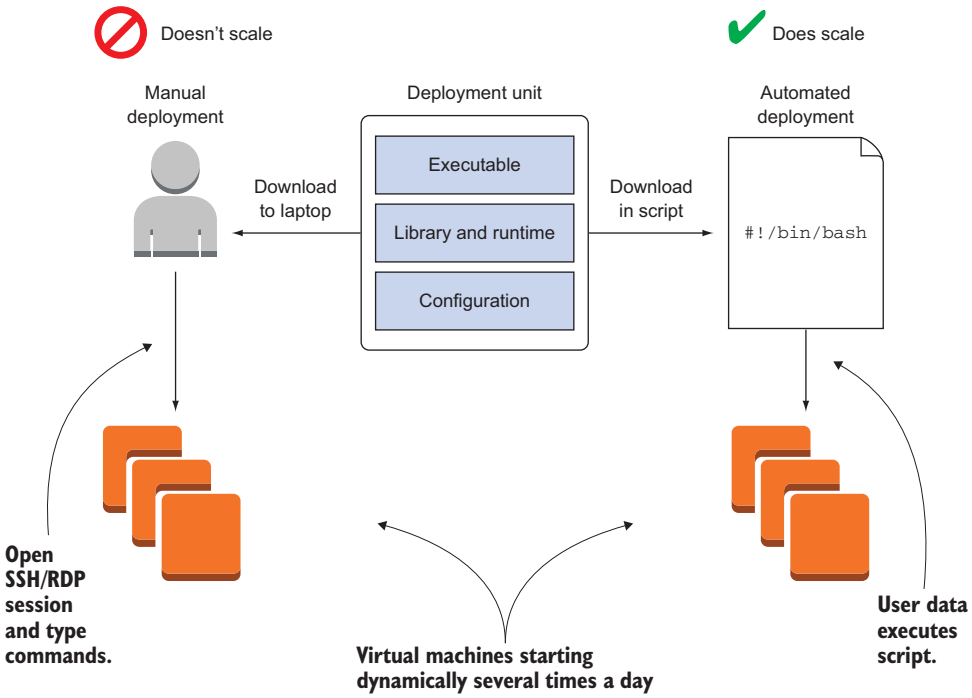


Figure 5.1 Deployment must be automated in a flexible and scalable cloud environment.

VMs you'll have to keep updated will grow. The steps required to deploy an application don't change, but as figure 5.1 shows, you need to perform them on multiple VMs. Deploying software manually to a growing number of VMs becomes impossible over time and has a high risk of human failure. This is why we recommend that you automate the deployment of applications.

The investment in an automated deployment process will pay off in the future by increasing efficiency and decreasing human error. In the next section, you will learn about options for automation that you will examine in more detail throughout the rest of the chapter.

5.2 Comparing deployment tools

You will learn about three ways to deploy an application in this chapter:

1. Creating a virtual machine and running a deployment script on startup with AWS CloudFormation.
2. Using AWS Elastic Beanstalk to deploy a common web application.
3. Using AWS OpsWorks to deploy a multilayer application.

In this section, we'll discuss the differences between these solutions. After that, you will take a dive deep into each.

5.2.1 Classifying the deployment tools

Figure 5.2 depicts the three AWS deployment options. The effort required to deploy an application using AWS Elastic Beanstalk is low. To benefit from this, your application has to fit into the conventions of AWS Elastic Beanstalk. For example, the application must run in one of the standardized runtime environments. If you're using OpsWorks Stacks, you'll have more freedom to adapt the service to your application's needs. For example, you can deploy different layers that depend on each other, or you can use a custom layer to deploy any application with the help of a *Chef recipe*; this takes extra effort but gives you additional freedom. On the other end of the spectrum you'll find CloudFormation and deploying applications with the help of a script that runs at the end of the boot process. You can deploy any application with the help of CloudFormation. The disadvantage is that you have to do more work, because you don't use standard tooling.

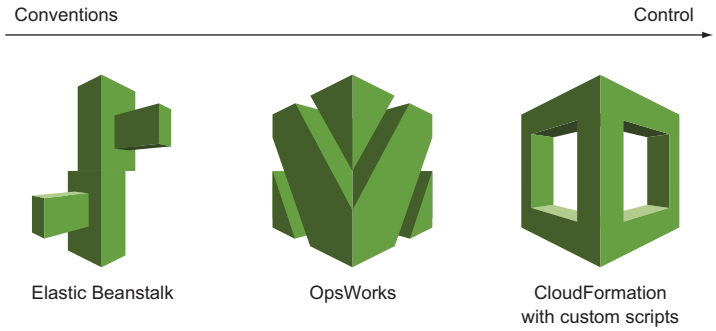


Figure 5.2 Comparing different ways to deploy applications on AWS

5.2.2 Comparing the deployment services

The classification in the previous section can help you decide the best fit to deploy an application. The comparison in table 5.1 highlights other important considerations.

Table 5.1 Differences between using CloudFormation with a script on virtual machine startup, Elastic Beanstalk, and OpsWorks Stacks

	CloudFormation with a script on VM startup	Elastic Beanstalk	OpsWorks
Configuration management tool	All available tools	Proprietary	Chef
Supported platforms	Any	▪ PHP ▪ Node.js ▪ .NET on Windows Server with IIS ▪ Java (SE or Tomcat) ▪ Python ▪ Ruby ▪ Go ▪ Docker	▪ PHP ▪ Node.js ▪ Java (Tomcat) ▪ Ruby on Rails ▪ Custom / Any

Table 5.1 Differences between using CloudFormation with a script on virtual machine startup, Elastic Beanstalk, and OpsWorks Stacks *(continued)*

	CloudFormation with a script on VM startup	Elastic Beanstalk	OpsWorks
Supported deployment artifacts	Anything	Zip archive on Amazon S3	Git, SVN, archive (such as Zip)
Common scenario	Medium- to enterprise-sized companies	Small companies	Companies with prior experience using Chef
Update without downtime	Not by default, but possible	Yes	Yes
Vendor lock-in effect	Medium	High	Medium

Many other options are available for deploying applications on AWS, from open source software to third-party services. Our advice is to use one of the AWS deployment services because they're well integrated into many other AWS services. We recommend that you use AWS CloudFormation with user data to deploy applications, because it's a flexible approach.

An automated deployment process will help you to iterate and innovate more quickly. You'll deploy new versions of your applications more often. To avoid service interruptions, you need to think about testing changes to software and infrastructure in an automated way, and being able to roll back to a previous version quickly if necessary.

In the next section you will use a Bash script and CloudFormation to deploy an application.

5.3 Creating a virtual machine and run a deployment script on startup with AWS CloudFormation

A simple but powerful and flexible way to automate application deployment is to launch a virtual machine and then launch a script at startup. To go from a plain OS to a fully installed and configured VM, follow these steps:

1. Start a plain virtual machine containing just an OS.
2. Execute a script the end of the boot process.
3. Install and configure your applications with the help of the script.

First you need to choose an AMI from which to start your virtual machine. An AMI bundles the OS for your VM with preinstalled software. When you start your VM from an AMI containing a plain OS without any additional software installed, you need to provision the VM at the end of the boot process. Otherwise all your virtual machines will look the same and will run a plain OS, which is not very helpful. You want to install custom applications, not a plain OS. Translating the necessary steps to install and configure your application into a script allows you to automate this task. But how do you execute this script automatically after booting your virtual machine?

5.3.1 Using user data to run a script on startup

You can inject a small amount of data called *user data*—no more than 16 KB— into every VM to customize them besides what comes in the AMI. You specify this user data during the creation of a new VM and can query it later from the machine itself. A typical way of using this feature is built into most AMIs, such as the Amazon Linux Image and the Ubuntu AMI. Whenever you boot a VM based on these AMIs, user data is executed as a shell script at the end of the boot process. The script is executed as the root user.

The user data is always accessible from the VM with a HTTP `GET` request to http://169.254.169.254/latest/user-data. The user data behind this URL is only accessible from the VM itself. As you'll see in the following example, you can deploy applications of any kind with the help of user data executed as a script.

5.3.2 Deploying OpenSwan: a VPN server to a virtual machine

If you're working over public Wi-Fi, for example using a laptop at a coffee house, you may want to tunnel your traffic through a VPN because unencrypted communication (such as HTTP instead of HTTPS) can be intercepted by an attacker. You'll learn how to deploy a VPN server to a virtual machine with the help of user data and a shell script next. The VPN solution, called *OpenSwan*, offers an IPSec-based tunnel that's easy to use with Windows, macOS, and Linux. Figure 5.3 shows the example setup.

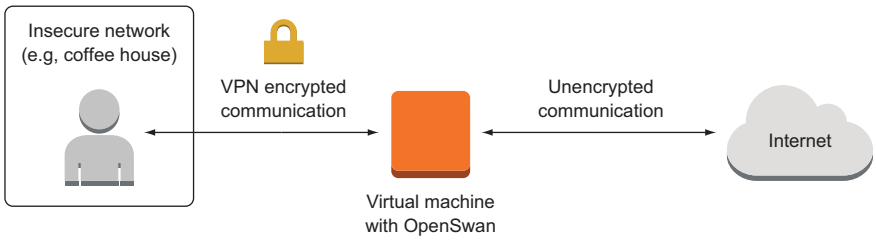


Figure 5.3 Using OpenSwan on a virtual machine to tunnel traffic from a personal computer

Open your terminal, and execute the commands shown in listing 5.1 step by step to start a virtual machine and deploy a VPN server on it. We've prepared a CloudFormation template that starts the virtual machine and its dependencies.

> **Shortcut for Linux and macOS**
>
> You can avoid typing these commands manually at your terminal by using the following command to download a Bash script and execute it directly on your local machine. The Bash script contains the same steps as shown in listing 5.1.
>
> ```
> $ curl -s https://raw.githubusercontent.com/AWSinAction/\
> ➥ code2/master/chapter05/\
> ➥ vpn-create-cloudformation-stack.sh | bash -ex
> ```

Listing 5.1 Deploying a VPN server to a virtual machine: CloudFormation and a shell script

```
$ VpcId="$(aws ec2 describe-vpcs --query "Vpcs[0].VpcId" --output text)"

$ SubnetId="$(aws ec2 describe-subnets --filters "Name=vpc-id,Values=$VpcId" \
➥ --query "Subnets[0].SubnetId" --output text)"

$ SharedSecret="$(openssl rand -base64 30)"

$ Password="$(openssl rand -base64 30)"

$ aws cloudformation create-stack  --stack-name vpn --template-url \
➥ https://s3.amazonaws.com/awsinaction-code2/chapter05/\
➥ vpn-cloudformation.yaml \
➥ --parameters ParameterKey=KeyName,ParameterValue=mykey \
➥ "ParameterKey=VPC,ParameterValue=$VpcId" \
➥ "ParameterKey=Subnet,ParameterValue=$SubnetId" \
➥ "ParameterKey=IPSecSharedSecret,ParameterValue=$SharedSecret" \
➥ ParameterKey=VPNUser,ParameterValue=vpn \
➥ "ParameterKey=VPNPassword,ParameterValue=$Password"

aws cloudformation wait stack-create-complete --stack-name vpn

$ aws cloudformation describe-stacks --stack-name vpn \
➥ --query "Stacks[0].Outputs"
```

Gets the default subnet

Gets the default VPC

Creates a random shared secret. (If openssl is not working, create your own secret.)

Creates a random password (if openssl is not working, create your own random sequence).

Creates a CloudFormation stack

Wait until stack is CREATE_COMPLETE.

Get stack outputs.

The output of the last command should print out the public IP address of the VPN server, a shared secret, the VPN username, and the VPN password. You can use this information to establish a VPN connection from your computer, if you like:

```
[{
  "Description": "The username for the vpn connection",
  "OutputKey": "VPNUser",
  "OutputValue": "vpn"
}, {
  "Description": "The shared key for the VPN connection (IPSec)",
  "OutputKey": "IPSecSharedSecret",
  "OutputValue": "EtAYOHXaLjcJ9nLCLEBfkZ+qV3H4Jy3MMc03Ehfy"
}, {
  "Description": "Public IP address of the virtual machine",
  "OutputKey": "ServerIP",
  "OutputValue": "34.202.233.247"
}, {
  "Description": "The password for the vpn connection",
  "OutputKey": "VPNPassword",
  "OutputValue": "MXBOtTlx3boJV+2r3tlOs6MCQisMhcj8oLVLilO2"
}]
```

Let's take a deeper look at the deployment process of the VPN server. We'll examine the following tasks, which you've already used:

- Starting a virtual machine with custom user data and configuring a firewall for the VM with AWS CloudFormation.
- Executing a shell script at the end of the boot process to install an application and its dependencies with the help of a package manager, as well as to edit configuration files

USING CLOUDFORMATION TO START A VIRTUAL MACHINE WITH USER DATA

You can use CloudFormation to start a virtual machine and configure a firewall. The template for the VPN server includes a shell script packed into user data, as shown in listing 5.2.

!Sub and !Base64

The CloudFormation template includes two new functions `!Sub` and `!Base64`. With `!Sub`, all references within `${}` are substituted with their real value. The real value will be the value returned by `!Ref`, unless the reference contains a dot, in which case it will be the value returned by `!GetAtt`:

```
!Sub 'Your VPC ID: ${VPC}' # becomes 'Your VPC ID: vpc-123456'
!Sub '${VPC}' # is the same as !Ref VPC
!Sub '${VPC.CidrBlock}' # is the same as !GetAtt 'VPC.CidrBlock'
!Sub '${!VPC}' # is the same as '${VPC}'
```

The function `!Base64` encodes the input with Base64. You'll need this function because the user data must be encoded in Base64:

```
!Base64 'value' # becomes 'dmFsdWU='
```

Listing 5.2 Parts of a CloudFormation template to start a virtual machine with user data

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 5 (OpenSwan acting as VPN IPSec endpoint)'
Parameters:                                          <-- Parameters to make it possible to reuse the template
  KeyName:
    Description: 'Key pair name for SSH access'
    Type: 'AWS::EC2::KeyPair::KeyName'
  VPC:
    Description: 'Just select the one and only default VPC.'
    Type: 'AWS::EC2::VPC::Id'
  Subnet:
    Description: 'Just select one of the available subnets.'
    Type: 'AWS::EC2::Subnet::Id'
  IPSecSharedSecret:
    Description: 'The shared secret key for IPSec.'
    Type: String
  VPNUser:
    Description: 'The VPN user.'
    Type: String
```

```
  VPNPassword:
    Description: 'The VPN password.'
    Type: String
Resources:
  EC2Instance:                                  # Describes the virtual machine
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: 't2.micro'
      KeyName: !Ref KeyName
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref InstanceSecurityGroup
        SubnetId: !Ref Subnet
      UserData:                                 # Defines a shell script as user data for the virtual machine
        'Fn::Base64': !Sub |                    # Substitutes and encodes a multi-line string value
          #!/bin/bash -x
          export IPSEC_PSK="${IPSecSharedSecret}"
          export VPN_USER="${VPNUser}"          # Exports parameters to environment variables to make them available in an external shell script called next
          export VPN_PASSWORD="${VPNPassword}"
          curl -s https://raw.githubusercontent.com/AWSinAction/code2/\
    ➥ master/chapter05/vpn-setup.sh | bash -ex   # Fetches the shell script via HTTP and executes it
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
    ➥ --resource EC2Instance --region ${AWS::Region}
      CreationPolicy:                           # CloudFormation will wait up to 10 minutes to receive a signal via the cfn-signal tool that runs in user data.
        ResourceSignal:
          Timeout: PT10M
  InstanceSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Enable access to VPN server'
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp                         # Signals end of script back to CloudFormation
        FromPort: 22
        ToPort: 22
        CidrIp: '0.0.0.0/0'
      - IpProtocol: udp
        FromPort: 500
        ToPort: 500
        CidrIp: '0.0.0.0/0'
      - IpProtocol: udp
        FromPort: 1701
        ToPort: 1701
        CidrIp: '0.0.0.0/0'
      - IpProtocol: udp
        FromPort: 4500
        ToPort: 4500
        CidrIp: '0.0.0.0/0'
Outputs:
  # [...]
```

The user data contains a small script to fetch and execute the real script, vpn-setup.sh, which contains all the commands for installing the executables and configuring the services. Doing so frees you from inserting complicated scripts in the CloudFormation template.

INSTALLING AND CONFIGURING A VPN SERVER WITH A SCRIPT

The vpn-setup.sh script shown in the following listing installs packages with the help of the package manager `yum` and writes some configuration files. You don't have to understand the details of the VPN server configuration; you only need to know that this shell script is executed during the boot process to install and configure a VPN server.

Listing 5.3 Installing packages and writing configuration files on virtual machine startup

```
#!/bin/bash -ex

[...]

PRIVATE_IP="$(curl -s http://169.254.169.254/latest/meta-data/local-ipv4)"

PUBLIC_IP="$(curl -s http://169.254.169.254/latest/meta-data/public-ipv4)"

yum-config-manager --enable epel
yum clean all

yum install -y openswan xl2tpd

cat > /etc/ipsec.conf <<EOF
[...]
EOF

cat > /etc/ipsec.secrets <<EOF
$PUBLIC_IP %any : PSK "${IPSEC_PSK}"
EOF

cat > /etc/xl2tpd/xl2tpd.conf <<EOF
[...]
EOF

cat > /etc/ppp/options.xl2tpd <<EOF
[...]
EOF

service ipsec start
service xl2tpd start

chkconfig ipsec on
chkconfig xl2tpd on
```

Fetches the private IP address of the virtual machine

Fetches the public IP address of the virtual machine

Adds extra packages to the package manager yum

Installs software packages

Writes a configuration file for IPSec (OpenSwan)

Writes a file containing the shared secret for IPSec

Writes a configuration file for the L2TP tunnel

Writes a configuration file for the PPP service

Starts the services needed for the VPN server

Configures the run level for the VPN services

That's it. You've now deployed a VPN server to a virtual machine with the help of EC2 user data and a shell script. If you want to test the VPN server, select the VPN type L2TP

over IPSec in your VPN client. After you terminate your virtual machine, you'll be ready to learn how to deploy a common web application without writing a custom script.

WARNING You've reached the end of the VPN server example. Don't forget to terminate your virtual machine and clean up your environment. To do so, enter `aws cloudformation delete-stack --stack-name vpn` at your terminal.

5.3.3 Starting from scratch instead of updating

You learned how to deploy an application with the help of user data in this section. The script from the user data is executed at the end of the boot process. But how do you update your application using this approach?

You've automated the installation and configuration of software during your VM's boot process, and can start a new VM without any extra effort. So if you have to update your application or its dependencies, it is easier to create a new up-to-date VM by following these steps:

1. Make sure an up-to-date version of your application or software is available through the package repository of your OS, or edit the user data script.
2. Start a new virtual machine based on your CloudFormation template and user data script.
3. Test the application deployed to the new virtual machine. Proceed with the next step if everything works as it should.
4. Switch your workload to the new virtual machine (for example, by updating a DNS record).
5. Terminate the old virtual machine, and throw away its unused dependencies.

5.4 Deploying a simple web application with AWS Elastic Beanstalk

If you have to deploy a common web application, you don't have to reinvent the wheel. AWS offers a service called *AWS Elastic Beanstalk* that can help you deploy web applications based on Go, Java (SE or Tomcat), .NET on Windows Server with IIS, Node.js, PHP, Python, Ruby, and Docker. With AWS Elastic Beanstalk, you don't have to worry about your OS or virtual machines. AWS will manage them for you (if you enable automatic updates). With Elastic Beanstalk, you only deal with your application. The OS and the runtime (such as Apache + Tomcat) are managed by AWS.

AWS Elastic Beanstalk lets you handle the following recurring problems:

- Providing a runtime environment for a web application (PHP, Java, and so on)
- Updating the runtime environment for a web application
- Installing and updating a web application automatically
- Configuring a web application and its environment
- Scaling a web application to balance load
- Monitoring and debugging a web application

5.4.1 Components of AWS Elastic Beanstalk

Getting to know the different components of AWS Elastic Beanstalk will help you to understand its functionality. Figure 5.4 shows these elements:

- An *application* is a logical container. It contains versions, environments, and configurations. If you start to use AWS Elastic Beanstalk in a region, you have to create an application first.
- A *version* contains a specific release of your application. To create a new version, you have to upload your executables (packed into an archive) to Amazon S3, which stores static files. A version is basically a pointer to this archive of executables.
- A *configuration template* contains your default configuration. You can manage your application's configuration (such as the port your application listens on) as well as the environment's configuration (such as the size of the virtual machine) with your custom configuration template.
- An *environment* is where AWS Elastic Beanstalk executes your application. It consists of a *version* and the *configuration*. You can run multiple environments for one application by using different combinations of versions and configurations.

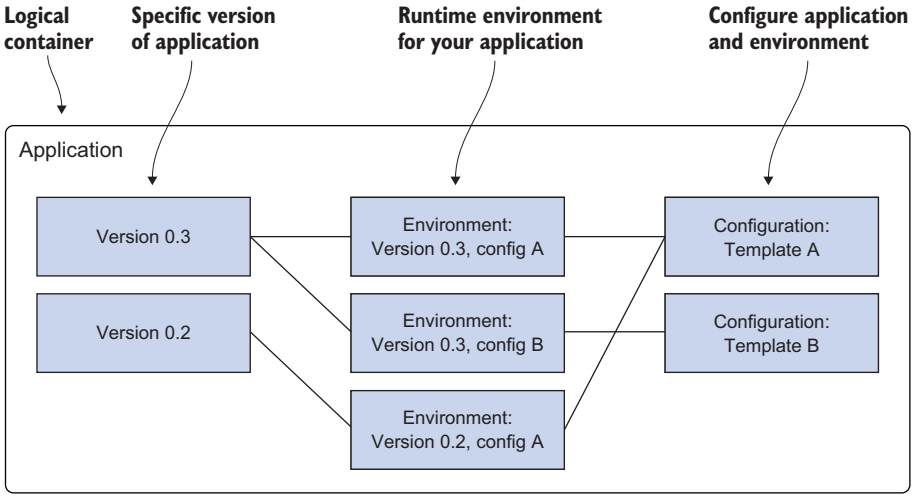


Figure 5.4 An AWS Elastic Beanstalk application consists of versions, environments, and configurations.

Enough theory for the moment. Let's proceed with deploying a simple web application.

5.4.2 Using AWS Elastic Beanstalk to deploy Etherpad, a Node.js application

Editing a document collaboratively can be painful if you're using the wrong tools. *Etherpad* is an open source online editor that lets you edit a document with many people in real time. You'll deploy this Node.js-based application with the help of AWS Elastic Beanstalk in three steps:

1. Create an application: the logical container.
2. Create a version: a pointer to a specific version of Etherpad.
3. Create an environment: the place where Etherpad will run.

Creating an application for AWS Elastic Beanstalk

Open your terminal, and execute the following command to create an application for the AWS Elastic Beanstalk service:

```
$ aws elasticbeanstalk create-application --application-name etherpad
```

You've now created a container for all the other components that are necessary to deploy Etherpad with the help of AWS Elastic Beanstalk.

Creating a version for AWS Elastic Beanstalk

You can create a new version of your Etherpad application with the following command:

```
$ aws elasticbeanstalk create-application-version --application-name etherpad \
➥ --version-label 1 \
➥ --source-bundle "S3Bucket=awsinaction-code2,S3Key=chapter05/etherpad.zip"
```

By executing this command, you created a version labeled `1`. For this example, we uploaded a zip archive containing Etherpad that you can use for convenience.

Creating an environment to execute Etherpad with AWS Elastic Beanstalk

To deploy Etherpad with the help of AWS Elastic Beanstalk, you have to create an environment for Node.js based on Amazon Linux and the version of Etherpad you just created. To get the latest Node.js environment version, called a *solution stack name*, run this command:

```
$ aws elasticbeanstalk list-available-solution-stacks --output text \
➥ --query "SolutionStacks[?contains(@, 'running Node.js')] | [0]"
64bit Amazon Linux 2017.03 v4.2.1 running Node.js
```

When AWS releases a new solution stack, this output may look different.

Execute the following command to launch an environment, replacing *`$SolutionStackName`* with the output from the previous command.

```
$ aws elasticbeanstalk create-environment --environment-name etherpad \
➥ --application-name etherpad \
➥ --option-settings Namespace=aws:elasticbeanstalk:environment,\
➥ OptionName=EnvironmentType,Value=SingleInstance \
➥ --solution-stack-name "$SolutionStackName" \
➥ --version-label 1
```

Launches a single virtual machine without the ability to scale and load-balance automatically

Having fun with Etherpad

You've now created an environment for Etherpad. It will take several minutes before you can point your browser to your Etherpad installation. The following command will help you track the state of your Etherpad environment:

```
$ aws elasticbeanstalk describe-environments --environment-names etherpad
```

When Status turns to Ready and Health turns to Green, you're ready to create your first Etherpad document. The output of the `describe` command should look similar to the following example.

Listing 5.4 Describing the status of the Elastic Beanstalk environment

```
{
  "Environments": [{
    "ApplicationName": "etherpad",
    "EnvironmentName": "etherpad",
    "VersionLabel": "1",
    "Status": "Ready",
    "EnvironmentLinks": [],
    "PlatformArn": "arn:aws:elasticbeanstalk:us-east-1::platform/Node.js
➥ running on 64bit Amazon Linux/4.2.1",
    "EndpointURL": "54.157.76.149",
    "SolutionStackName": "64bit Amazon Linux 2017.03 v4.2.1 running Node.js",
    "EnvironmentId": "e-8d532q3vkk",
    "CNAME": "etherpad.d2nhjs7myw.us-east-1.elasticbeanstalk.com",
    "AbortableOperationInProgress": false,
    "Tier": {
      "Version": " ",
      "Type": "Standard",
      "Name": "WebServer"
    },
    "Health": "Green",
    "DateUpdated": "2017-08-15T09:18:47.750Z",
    "DateCreated": "2017-08-15T09:14:32.137Z"
  }]
}
```

Wait until Status turns to Ready.

DNS record for the environment (for example, to open with a browser)

Wait until Health turns to Green.

You've now deployed a Node.js web application to AWS in three simple steps. Point your browser to the URL shown in `CNAME`, and open a new document by typing in a name for it and clicking OK. If the page does not load, try the `EndpointURL`, which is a public IP address. The CNAME should work within the next few minutes as well. Figure 5.5 shows an Etherpad document in action.

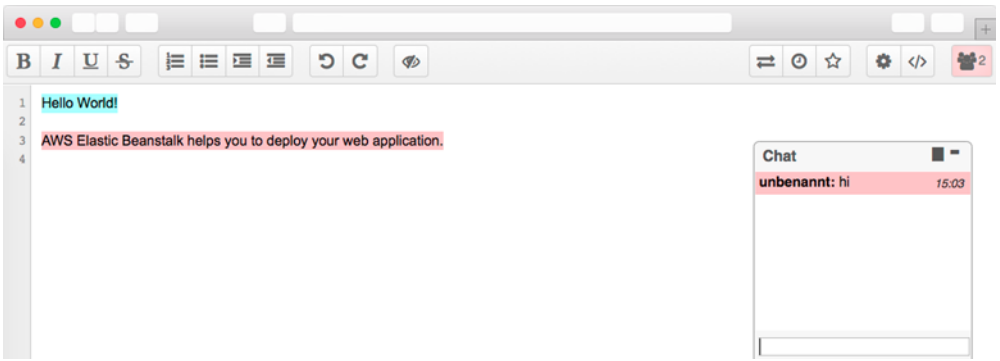


Figure 5.5 Online text editor Etherpad in action

If you want to deploy any other Node.js application, the only thing that changes is the zip file that you upload to Elastic Beanstalk. If you want to run something other than a Node.js application, you have to use the appropriate solution stack name with `aws elasticbeanstalk list-available-solution-stacks`.

Exploring AWS Elastic Beanstalk with the Management Console

You've deployed Etherpad using AWS Elastic Beanstalk and the AWS CLI by creating an application, a version, and an environment. You can also control AWS Elastic Beanstalk using the web-based Management Console. In our experience, the Management Console is the best way to manage AWS Elastic Beanstalk.

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Click Services in the navigation bar, and click the Elastic Beanstalk service.
3. Click the etherpad environment, represented by a green box. An overview of the Etherpad application is shown, as in figure 5.6.

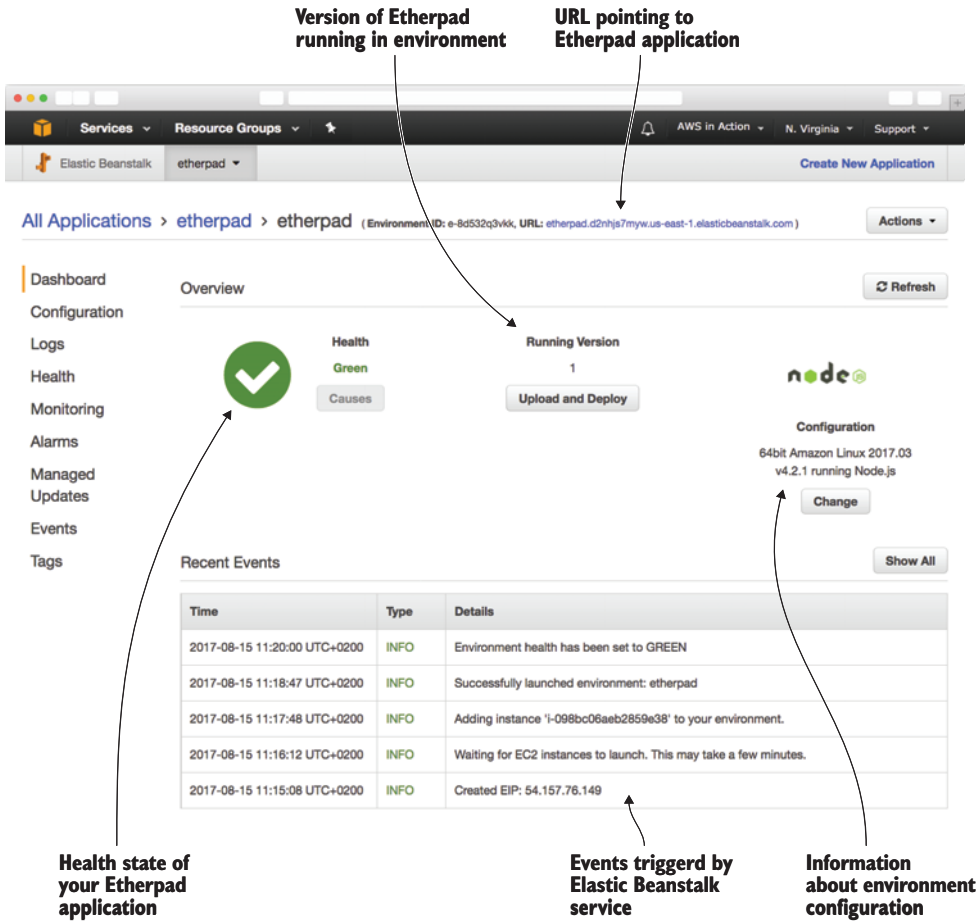


Figure 5.6 Overview of AWS Elastic Beanstalk environment running Etherpad

What if something goes wrong with your application? How can you debug an issue? Usually you connect to the virtual machine and look at the log messages. You can fetch the log messages from your application (and other components) using AWS Elastic Beanstalk. Follow these steps:

1. Choose Logs from the submenu. You'll see a screen like that shown in figure 5.7.
2. Click Request Logs, and choose Last 100 Lines.
3. After a few seconds, a new entry will appear in the table. Click Download to download the log file to your computer.

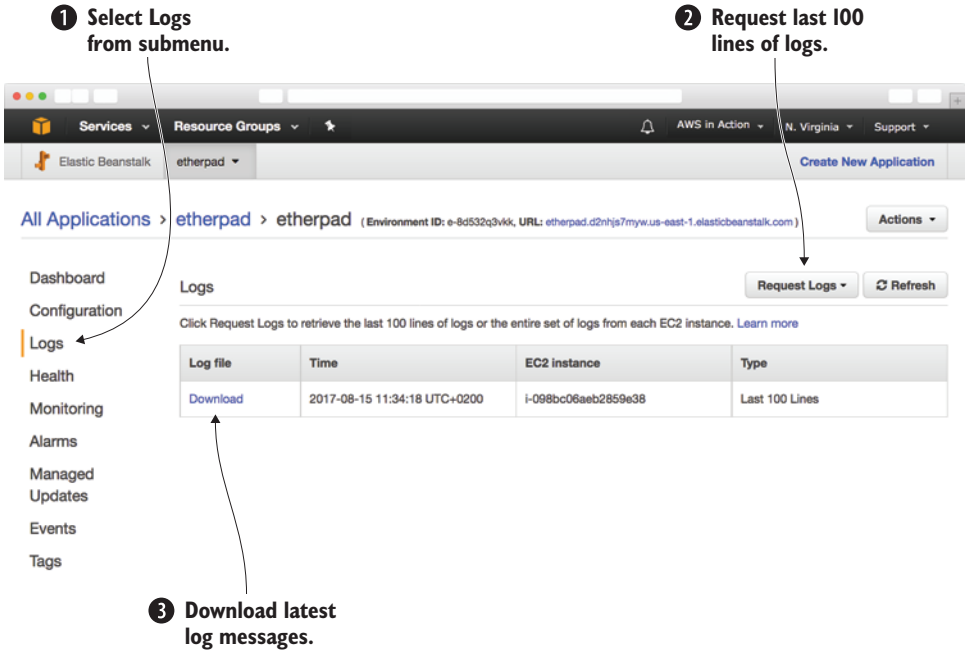


Figure 5.7 Downloading logs from a Node.js application via AWS Elastic Beanstalk

Cleaning up AWS Elastic Beanstalk

Now that you've successfully deployed Etherpad using AWS Elastic Beanstalk and learned about the service's different components, it's time to clean up. Run the following command to terminate the Etherpad environment:

```
$ aws elasticbeanstalk terminate-environment --environment-name etherpad
```

You can check the state of the environment by executing the following command:

```
$ aws elasticbeanstalk describe-environments --environment-names etherpad \
➥ --output text --query "Environments[].Status"
```

Wait until Status has changed to Terminated, and then proceed with the following command:

```
$ aws elasticbeanstalk delete-application --application-name etherpad
```

That's it. You've terminated the virtual machine providing the environment for Etherpad and deleted all components of AWS Elastic Beanstalk.

5.5 Deploying a multilayer application with AWS OpsWorks Stacks

Deploying a basic web application using AWS Elastic Beanstalk is convenient. But if you have to deploy a more complex application consisting of different services—also called *layers*—you'll reach the limits of AWS Elastic Beanstalk. In this section, you'll learn about AWS OpsWorks Stacks, a free service offered by AWS that can help you to deploy a multilayer application.

AWS OpsWorks flavors

AWS OpsWorks comes in different two flavors:

- *AWS OpsWorks Stacks* comes with Chef versions 11 and 12. In Chef 11, OpsWorks comes with a bunch of built-in layers, which is best for beginners. If you have Chef knowledge, this may limit you. We recommend to use OpsWorks Stacks with Chef 12 if you have Chef knowledge, because there are no limiting built-in layers.
- *AWS OpsWorks for Chef Automate* provides a Chef Automate server and takes care about backups, restorations, and software upgrades. You should use OpsWorks for Chef Automate if you have an existing infrastructure managed by Chef that you want to migrate to AWS.

AWS OpsWorks Stacks helps you control AWS resources like virtual machines, load balancers, container clusters, and databases, and lets you deploy applications. The service offers some standard layers with the following runtimes:

- HAProxy (load balancer)
- Static web server
- Rails app server (Ruby on Rails)
- PHP app server
- Node.js app server
- Java app server (Tomcat server)
- AWS Flow (Ruby)
- MySQL (database)
- Memcached (in-memory cache)
- Ganglia (monitoring)

You can also add a custom layer to deploy anything you want. The deployment is controlled with the help of *Chef*, a configuration management tool. Chef uses *recipes* organized into *cookbooks* to deploy applications to any kind of system. You can adopt the standard recipes or create your own.

About Chef

Chef is a configuration management tool similar to Puppet, SaltStack, CFEngine, and Ansible. Chef lets you configure and deploy applications by transforming templates (recipes) written in a domain-specific language (DSL) into actions. A recipe can include packages to install, services to run, or configuration files to write, for example. Related recipes can be combined into cookbooks. Chef analyzes the status quo and changes resources where necessary to reach the described state from the recipe.

You can reuse cookbooks and use recipes you get from others. The community publishes a variety of cookbooks and recipes at https://supermarket.chef.io under open source licenses.

Chef can be run in solo or client/server mode. It acts as a fleet-management tool in client/server mode. This can help if you have to manage a distributed system consisting of many VMs. In solo mode, you can execute recipes on a single VM. AWS OpsWorks uses solo mode integrated into its own fleet management, without requiring you to configure and operate a setup in client/server mode.

Besides deploying your application, AWS OpsWorks Stacks can help you to scale, monitor, and update your VMs running beneath the different layers.

5.5.1 Components of AWS OpsWorks Stacks

Getting to know the different components of AWS OpsWorks Stacks will help you understand its functionality. Figure 5.8 shows these elements:

- A *stack* is a container for all other components of AWS OpsWorks Stacks. You can create one or more stacks and add one or more layers to each stack. You could use different stacks to separate the production environment from the testing environment, for example. Or you could use different stacks to separate different applications.
- A *layer* belongs to a stack. A layer represents an application; you could also call it a service. AWS OpsWorks Stacks offers predefined layers for standard web applications like PHP and Java, but you're free to use a custom stack for any application you can think of. Layers are responsible for configuring and deploying software to virtual machines. You can add one or multiple VMs to a layer; in this context the VMs are called *instances*.
- An *instance* is the representation for a virtual machine. You can launch one or multiple instances for each layer, using different versions of Amazon Linux and

Ubuntu or a custom AMI as a basis for the instances. You can specify rules for launching and terminating instances based on load or timeframes for scaling.

- An *app* is the software you want to deploy. AWS OpsWorks Stacks deploys your app to a suitable layer automatically. You can fetch apps from a Git or Subversion repository, or as archives via HTTP. AWS OpsWorks Stacks helps you to install and update your apps onto one or multiple instances.

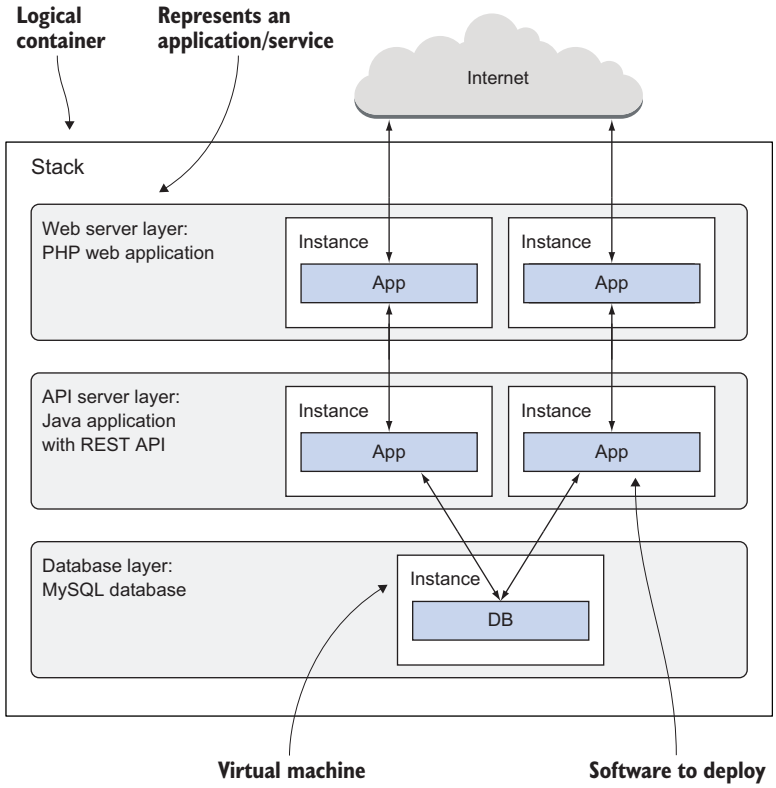


Figure 5.8 Stacks, layers, instances, and apps are the main components of AWS OpsWorks Stacks.

Let's look at how to deploy a multilayer application with the help of AWS OpsWorks Stacks.

5.5.2 Using AWS OpsWorks Stacks to deploy an IRC chat application

Internet Relay Chat (IRC) is still a popular means of communication in some circles. In this section, you'll deploy *kiwiIRC*, a web-based IRC client, and your own IRC server. Figure 5.9 shows the setup of a distributed system consisting of a web application delivering the IRC client, as well as an IRC server.

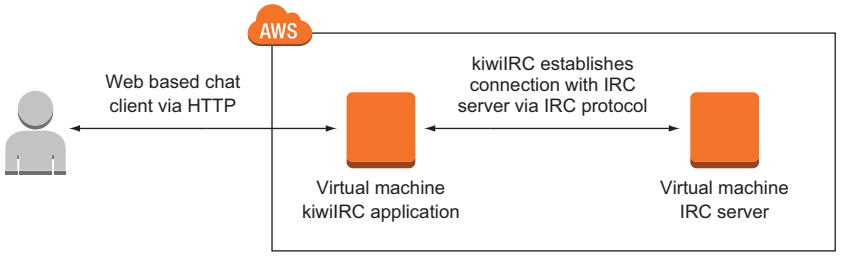


Figure 5.9 Building your own IRC infrastructure consisting of a web application and an IRC server

kiwiIRC is an open source web application written in JavaScript for Node.js. To deploy it as a two-layer application using AWS OpsWorks Stacks, you need scripts that do the following:

1. Create a stack, the container for all other components.
2. Create a Node.js layer for kiwiIRC.
3. Create a custom layer for the IRC server.
4. Create an app to deploy kiwiIRC to the Node.js layer.
5. Add an instance for each layer.

You'll learn how to handle these steps with the Management Console. You can also control AWS OpsWorks Stacks with the CLI, just like AWS Elastic Beanstalk or AWS CloudFormation.

CREATING A NEW OPSWORKS STACK

Open the Management Console at https://console.aws.amazon.com/opsworks, and click the Go to OpsWorks Stacks button. There you can start fresh by adding a new stack. Figure 5.10 illustrates the necessary steps with the most important highlighted:

1. Click Add stack under Select Stack or Add Your First Stack.
2. Select Chef 11 stack.
3. For Name, type in `irc`.
4. For Region, choose US East (N. Virginia).
5. The default VPC is the only one available. Select it.
6. For Default subnet, select us-east-1a.
7. For Default operating system, choose Ubuntu 14.04 LTS.
8. Select your SSH key, mykey, for Default SSH key.
9. Click Add stack to create the stack.

You're now redirected to an overview of your IRC AWS OpsWorks stack. Everything is ready for you to create the first layer.

Figure 5.10 Creating a stack with AWS OpsWorks Stacks

Creating a Node.js layer for an OpsWorks stack

kiwiIRC is a Node.js application, so you need to create a Node.js layer for the IRC stack. Follow these steps in figure 5.11 to do so:

1. Select Layers from the submenu on the left.
2. Click the Add layer button.
3. For Layer type, select Node.js App Server.
4. Select the latest 0.12.x version of Node.js.
5. Click Add layer.

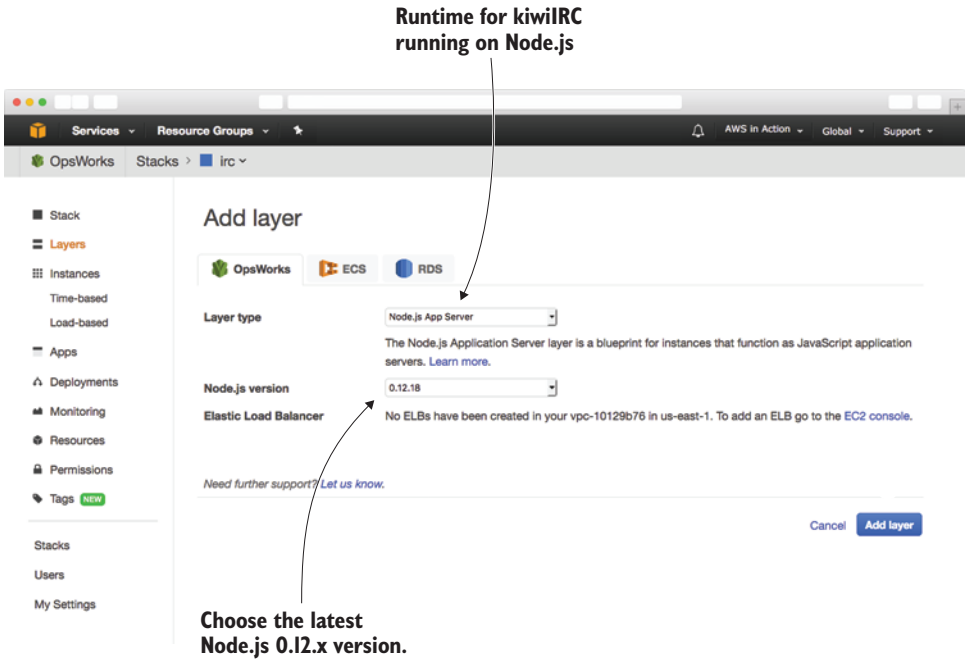


Figure 5.11 Creating a layer with Node.js for kiwiIRC

You've created a Node.js layer. Now you need to repeat these steps to add another layer and deploy your own IRC server.

CREATING A CUSTOM LAYER FOR AN OPSWORKS STACK

An IRC server isn't a typical web application, so the default layer types are out of the question. You'll use a custom layer for deploying an IRC server. The Ubuntu package repository includes various IRC server implementations; you'll use the `ircd-ircu` package. Follow these steps in figure 5.12 to create a custom stack for the IRC server:

1. Select Layers from the submenu on the left.
2. Click Add layer.
3. For Layer type, select Custom.
4. For Name and for Short name, type in `irc-server`.
5. Click Add layer.

You've now created a custom layer. If you want to deploy any other application, go with one of the pre-built layers first. If that is not possible, use a custom layer. This way, you benefit best from OpsWorks.

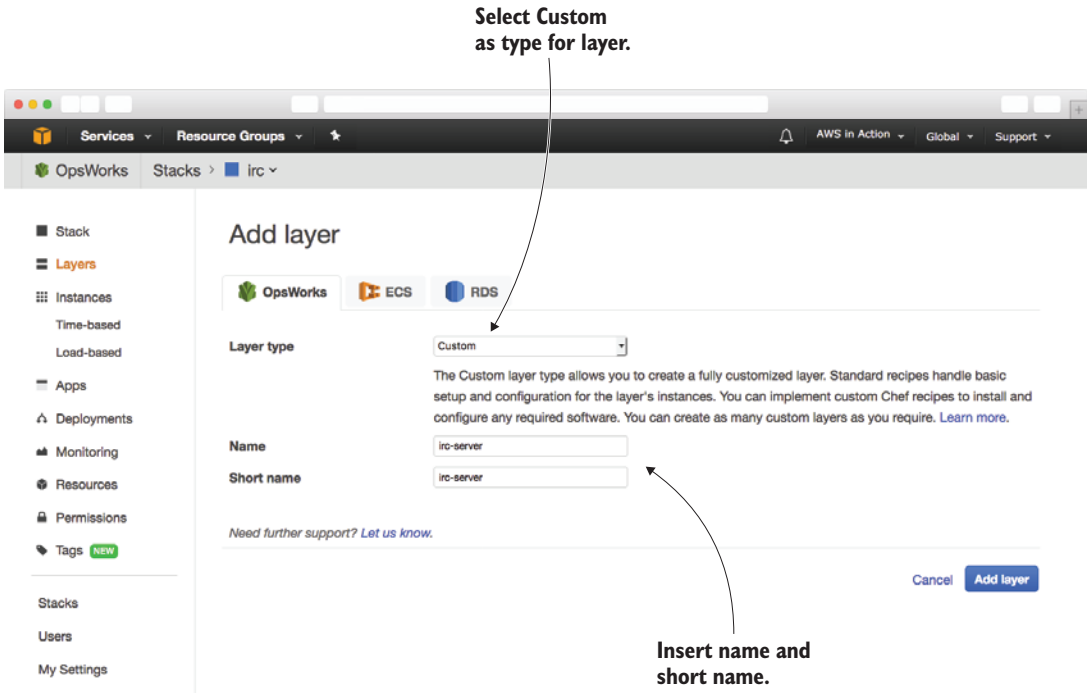


Figure 5.12 Creating a custom layer to deploy an IRC server

The IRC server needs to be reachable through port 6667. To allow access to this port, you need to define a custom firewall. Execute the commands shown in listing 5.5 to create a custom firewall for your IRC server.

Shortcut for Linux and macOS

You can avoid typing these commands manually into your terminal by using the following command to download a Bash script and execute it directly on your local machine. The Bash script contains the same steps as shown in listing 5.5:

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
➥ code2/master/chapter05/irc-create-cloudformation-stack.sh \
➥ | bash -ex
```

Listing 5.5 Creating a custom firewall with the help of CloudFormation

```
$ VpcId="$(aws ec2 describe-vpcs --query "Vpcs[0].VpcId" --output text)"

$ aws cloudformation create-stack --stack-name irc \
➥ --template-url https://s3.amazonaws.com/awsinaction-code2/\
➥ chapter05/irc-cloudformation.yaml \
```

Gets the default VPC

Creates a CloudFormation stack

```
➥ --parameters "ParameterKey=VPC,ParameterValue=$VpcId"

$ aws cloudformation wait stack-create-complete --stack-name irc
```

Wait until stack is CREATE_COMPLETE.

Next you need to attach this custom firewall configuration to the custom OpsWorks layer. Follow these steps in figure 5.13:

1. Select Layers from the submenu on the left.
2. Open the irc-server layer by clicking it.
3. Change to the Security tab, and click Edit.
4. For custom Security groups, select the security group that starts with irc.
5. Click Save.

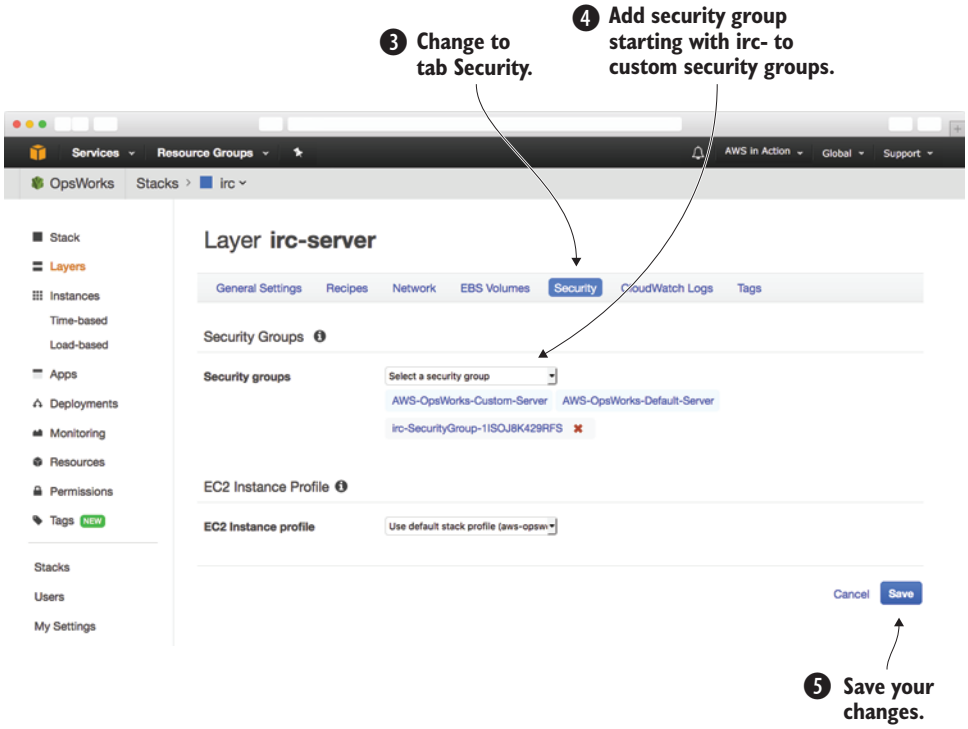


Figure 5.13 Adding a custom firewall configuration to the IRC server layer

You need to configure one last thing for the IRC server layer: the layer recipes for deploying an IRC server. Follow these steps in figure 5.14 to do so:

1. Select Layers from the submenu on the left.
2. Open the irc-server layer by clicking it.
3. Change to the Recipes tab, and click Edit.

4 For OS Packages, add the package `ircd-ircu`. Don't forget to click the + button to add the package.
5 Click save.

You've successfully created and configured a custom layer to deploy the IRC server. Next you'll add the kiwiIRC web application as an app to OpsWorks.

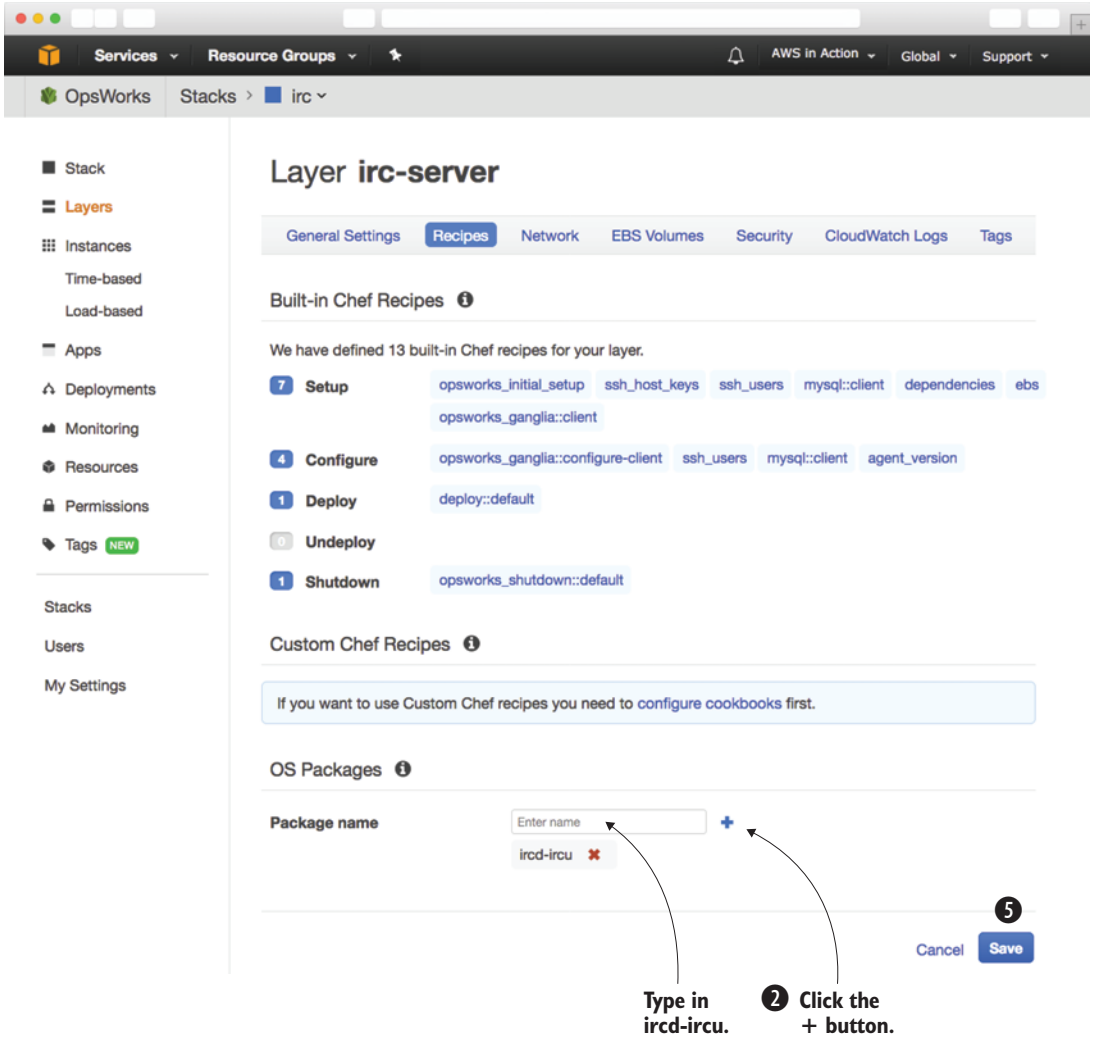


Figure 5.14 Adding an IRC package to a custom layer

ADDING AN APP TO THE NODE.JS LAYER

Now you're ready to deploy an app to the Node.js layer you just created. Follow these steps in figure 5.15:

1. Select Apps from the submenu.
2. Click the Add app button.
3. For Name, type in `kiwiIRC`.
4. For Type, select Node.js.
5. For Repository type, select Git, and type in `https://github.com/AWSinAction/KiwiIRC.git` for Repository URL.
6. Click the Add App button.

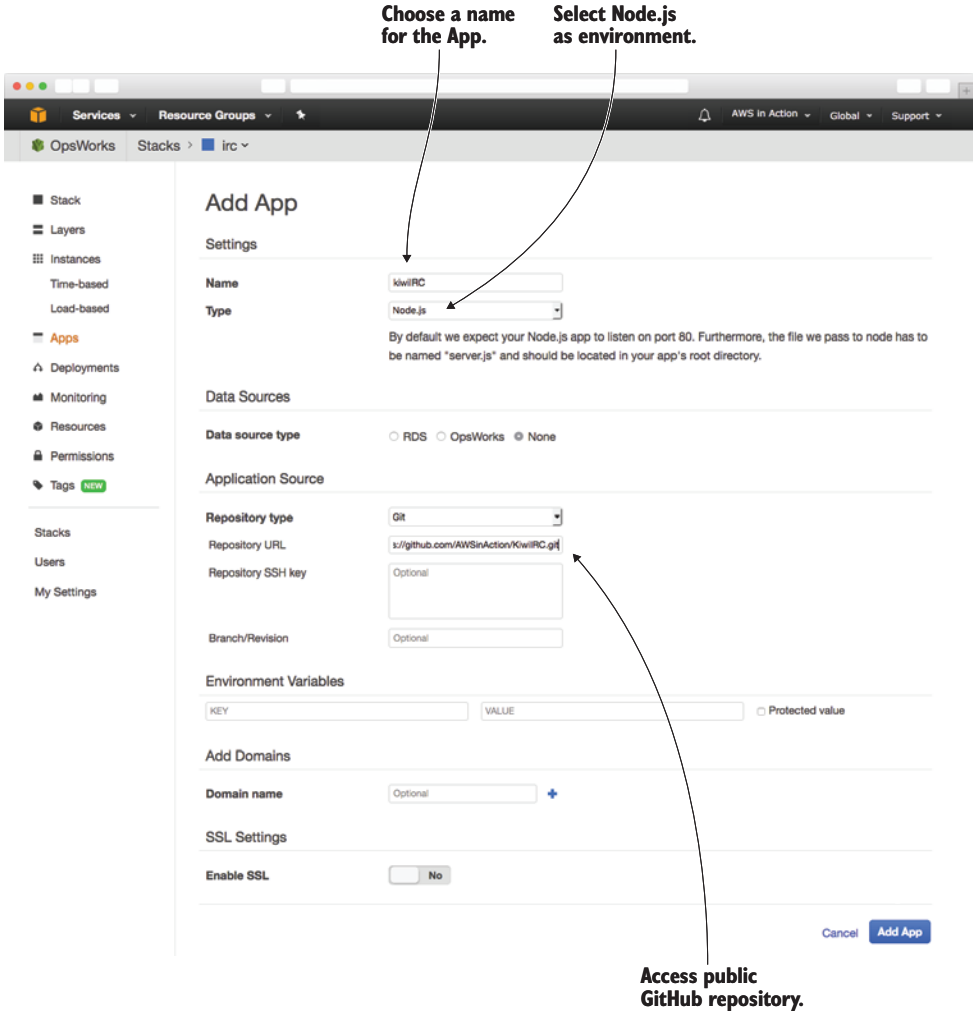


Figure 5.15 Adding kiwiIRC, a Node.js app, to OpsWorks

Your first OpsWorks stack is now fully configured. Only one thing is missing: you need to start some instances.

Adding instances to run the IRC client and server

Adding two instances will bring the kiwiIRC client and the IRC server into being. Adding a new instance to a layer is easy—follow these steps shown in figure 5.16:

1. Select Instances from the submenu on the left.
2. Click the Add instance button on the Node.js App Server layer.
3. For Size, select t2.micro, the instance type covered by the Free Tier.
4. Click Add instance.

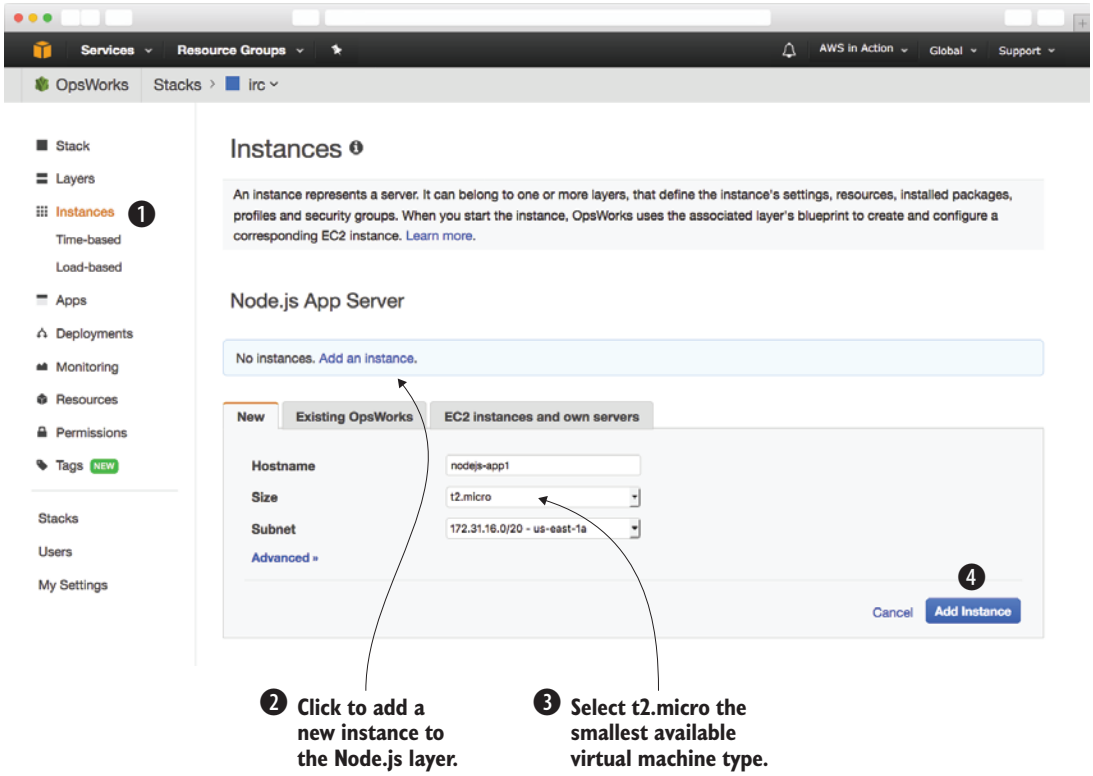


Figure 5.16 Adding a new instance to the Node.js layer

You've added an instance to the Node.js App Server layer. Repeat these steps for the irc-server layer as well.

The overview of instances should be similar to figure 5.17. To start them, click Start for both instances. It will take some time for the virtual machines to boot and the deployment to run, so this is a good time to get some coffee or tea.

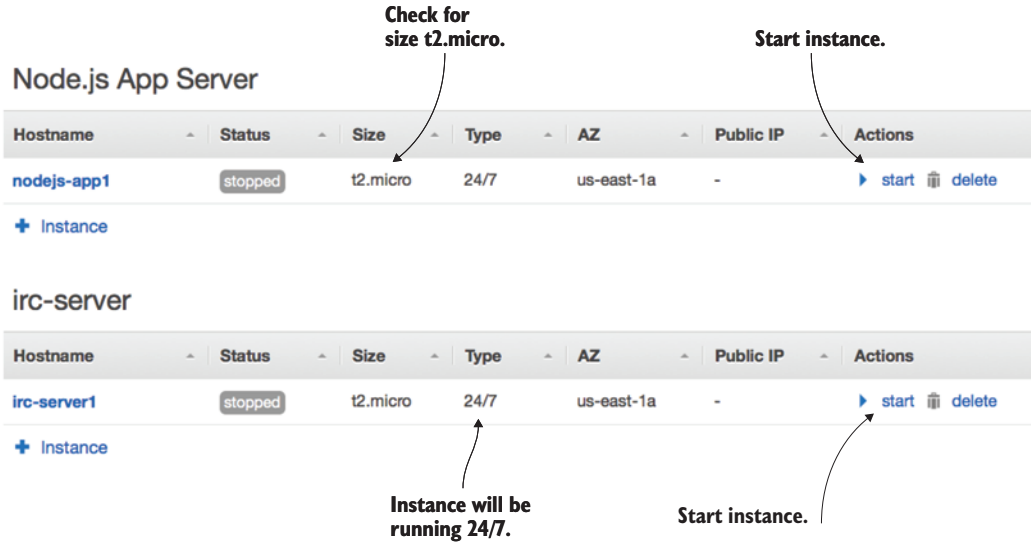


Figure 5.17 Starting the instances for the IRC web client and server

HAVING FUN WITH KIWIIRC

Be patient and wait until the status of both instances changes to Online, as shown in figure 5.18. You can now open kiwiIRC in your browser by following these steps:

1. Remember (or write down) the public IP address of the instance irc-server1. You'll need it to connect to your IRC server later.
2. Click the public IP address of the nodejs-app1 instance to open kiwiIRC in a new browser tab.

Figure 5.18 Waiting for deployment to open kiwiIRC in the browser

The kiwiIRC application should load in your browser, and you should see a login screen like the one in figure 5.19. Follow these steps to log in to your IRC server with the kiwiIRC web client:

1. Type in a nickname.
2. For Channel, type in `#awsinaction`.
3. Open the details of the connection by clicking Server and network.
4. Type the IP address of irc-server1 into the Server field.
5. For Port, type in `6667`.
6. Disable SSL.
7. Click Start, and wait a few seconds.

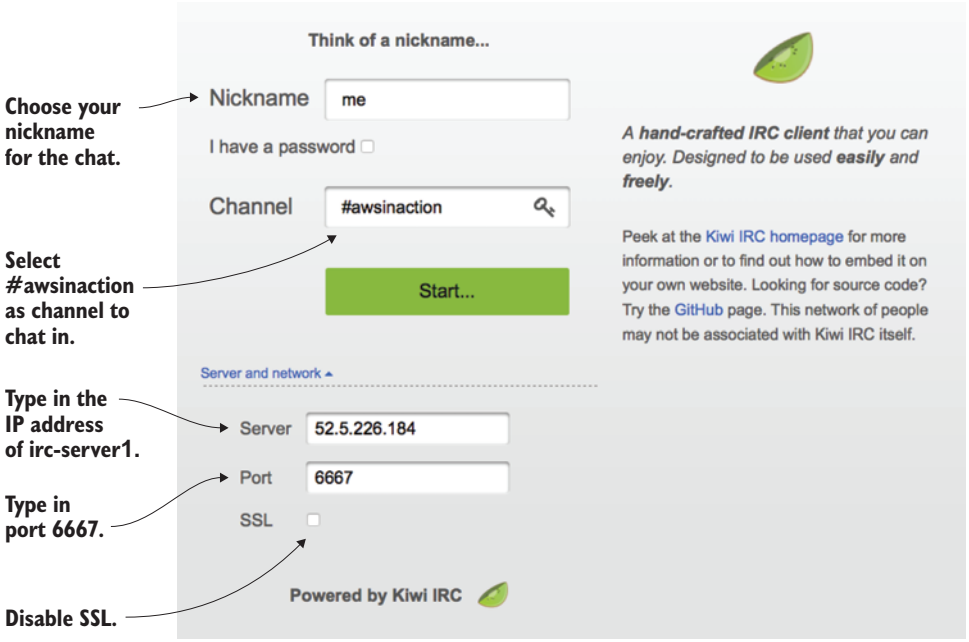


Figure 5.19 Using kiwiIRC to log in to your IRC server on channel #awsinaction

Congratulations! You've deployed a web-based IRC client and an IRC server with the help of AWS OpsWorks.

Cleaning up AWS OpsWorks

It's time to clean up. Follow these steps to avoid being charged unintentionally:

1. Open the AWS OpsWorks Stacks service with the Management Console.
2. Select the irc stack by clicking it.
3. Select Instances from the submenu..

(continued)

4. Delete both instances, and wait until they disappear from the overview.
5. Select Apps from the submenu.
6. Delete the kiwiIRC app.
7. Select Stack from the submenu.
8. Click the Delete Stack button, and confirm the deletion.
9. Execute `aws cloudformation delete-stack --stack-name irc` from your terminal.

Summary

- Automating the deployment of your applications onto virtual machines allows you to take full advantage of the cloud: scalability and high availability.
- AWS offers different tools for deploying applications onto virtual machines. Using one of these tools prevents you from reinventing the wheel.
- You can update an application by throwing away old VMs and starting new, up-to-date ones if you've automated your deployment process.
- Injecting Bash or PowerShell scripts into a virtual machine during startup allows you to initialize virtual machines individually—for example for installing software or configuring services.
- AWS OpsWorks is good for deploying multilayer applications with the help of Chef.
- AWS Elastic Beanstalk is best suited for deploying common web applications.
- AWS CloudFormation gives you the most control when you're deploying more complex applications.

6 Securing your system: IAM, security groups, and VPC

This chapter covers

- Who is responsible for security?
- Keeping your software up to date
- Controlling access to your AWS account with users and roles
- Keeping your traffic under control with security groups
- Using CloudFormation to create a private network

If security is a wall, you'll need a lot of bricks to build that wall as shown in figure 6.1. This chapter focuses on the four most important bricks to secure your systems on AWS:

1. *Installing software updates*—New security vulnerabilities are found in software every day. Software vendors release updates to fix those vulnerabilities and it's your job to install those updates as quickly as possible after they're released. Otherwise your system will be an easy victim for hackers.

2. *Restricting access to your AWS account*—This becomes even more important if you aren't the only one accessing your AWS account (if coworkers and scripts are also accessing it). A buggy script could easily terminate all your EC2 instances instead of only the one you intended. Granting only the permissions you need is key to securing your AWS resources from accidental or intended disastrous actions.
3. *Controlling network traffic to and from your EC2 instances*—You only want ports to be accessible if they must be. If you run a web server, the only ports you need to open to the outside world are port 80 for HTTP traffic and 443 for HTTPS traffic. Close down all the other ports!
4. *Creating a private network in AWS*—You can create subnets that aren't reachable from the internet. And if they're not reachable, nobody can access them. Really, nobody? You'll learn how you can get access to them while preventing others from doing so.

One important brick is missing: securing your applications. We do not cover application security in our book. When buying or developing applications, you should follow security standards. For example, you need to check user input and allow only the necessary characters, don't save passwords in plain text, and use TLS/SSL to encrypt traffic between your virtual machines and your users. If you are installing applications with a package manager for your operating system, using Amazon Inspector (https://aws.amazon.com/inspector/) allows you to run automated security assessments.

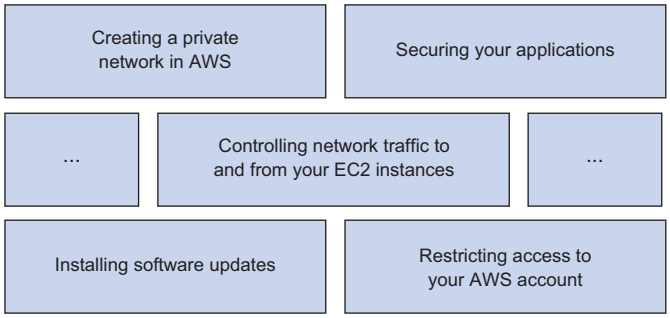


Figure 6.1 To achieve security of your cloud infrastructure and application, all security building blocks have to be in place.

Not all examples are covered by Free Tier

The examples in this chapter are *not* all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

Chapter requirements

To fully understand this chapter, you should be familiar with the following concepts:

- Subnet
- Route tables
- Access control lists (ACLs)
- Gateway
- Firewall
- Port
- Access management
- Basics of the Internet Protocol (IP), including IP addresses

Before we look at the four bricks, let's talk about how responsibility is divided between you and AWS.

6.1 Who's responsible for security?

The cloud is a shared-responsibility environment, meaning responsibility is shared between you and AWS. AWS is responsible for the following:

- Protecting the network through automated monitoring systems and robust internet access, to prevent Distributed Denial of Service (DDoS) attacks.
- Performing background checks on employees who have access to sensitive areas.
- Decommissioning storage devices by physically destroying them after end of life.
- Ensuring the physical and environmental security of data centers, including fire protection and security staff.

The security standards are reviewed by third parties; you can find an up-to-date overview at http://aws.amazon.com/compliance/.

What are your responsibilities?

- Implementing access management that restricts access to AWS resources like S3 and EC2 to a minimum, using AWS IAM.
- Encrypting network traffic to prevent attackers from reading or manipulating data (for example, using HTTPS).
- Configuring a firewall for your virtual network that controls incoming and outgoing traffic with security groups and ACLs.
- Encrypting data at rest. For example, enable data encryption for your database or other storage systems.
- Managing patches for the OS and additional software on virtual machines.

Security involves an interaction between AWS and you, the customer. If you play by the rules, you can achieve high security standards in the cloud.

6.2 Keeping your software up to date

Not a week goes by without the release of an important update to fix security vulnerabilities in some piece of software or another. Sometimes your OS is affected, or software libraries like OpenSSL. Other times it's an environments like Java, Apache, and PHP; or an application like WordPress. If a security update is released, you must install it quickly, because the exploit may have already been released, or because unscrupulous people could look at the source code to reconstruct the vulnerability. You should have a working plan for how to apply updates to all running virtual machines as quickly as possible.

6.2.1 Checking for security updates

If you log in to an Amazon Linux EC2 instance via SSH, you'll see message like the following:

```
$ ssh ec2-user@ec2-34-230-84-110.compute-1.amazonaws.com

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2017.03-release-notes/
8 package(s) needed for security, out of 8 available
Run "sudo yum update" to apply all updates.
```

Eight security updates are available.

This example shows that eight security updates are available; this number will vary when you look for updates. AWS won't apply updates for you on your EC2 instances—you're responsible for doing so.

You can use the `yum` package manager to handle updates on Amazon Linux. Run `yum --security check-update` to see which packages require a security update:

The output will be different when you run the command.

```
$ yum --security check-update
Loaded plugins: priorities, update-motd, upgrade-helper
8 package(s) needed for security, out of 8 available

authconfig.x86_64      6.2.8-30.31.amzn1    amzn-updates
bash.x86_64            4.2.46-28.37.amzn1   amzn-updates
curl.x86_64            7.51.0-9.75.amzn1    amzn-updates
glibc.x86_64           2.17-196.172.amzn1   amzn-updates
glibc-common.x86_64    2.17-196.172.amzn1   amzn-updates
kernel.x86_64          4.9.43-17.38.amzn1   amzn-updates
libcurl.x86_64         7.51.0-9.75.amzn1    amzn-updates
wget.x86_64            1.18-3.27.amzn1      amzn-updates
```

These packages are installed by default. Updates are available fixing security issues.

We encourage you to subscribe to the feed at the Amazon Linux AMI Security Center at https://alas.aws.amazon.com to receive security bulletins affecting Amazon Linux. Whenever a new security update is released, you should check whether you're affected

and act accordingly. If you are using another Linux distribution or operating system, you should follow the relevant security bulletins.

When dealing with security updates, you may face either of these situations:

- When the virtual machine starts the first time, many security updates may need to be installed in order for the machine to be up to date.
- New security updates are released when your virtual machine is running, and you need to install these updates while the machine is running.

Let's look how to handle these situations.

6.2.2 Installing security updates on startup

If you create your EC2 instances with CloudFormation templates, you have three options to install security updates on startup:

1. *Install all updates at the end of the boot process* by including `yum -y update` in your user-data script.
2. *Install security updates at the end of the boot process only* by including `yum -y --security update` in your user-data script.
3. *Define the package versions explicitly*. Install updates identified by a version number.

The first two options can be easily included in the user data of your EC2 instance. You can find the code in `/chapter06/ec2-yum-update.yaml` in the book's code folder. You install all updates as follows:

```
Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      # [...]
      UserData:
        'Fn::Base64': |
          #!/bin/bash -x
          yum -y update      <-- Installs all updates
```

To install only security updates, do the following:

```
Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      # [...]
      UserData:
        'Fn::Base64': |
          #!/bin/bash -x
          yum -y --security update        <-- Installs only security updates
```

The problem with installing all updates is that your system becomes unpredictable. If your virtual machine was started last week, all updates that were available last week have been applied. But in the meantime, new updates have likely been released. If you start a new VM today and install all updates, you'll end up with a different machine

than the machine from last week. *Different* can mean that for some reason it's no longer working. That's why we encourage you to explicitly define and test the updates you want to install. To install security updates with an explicit version, you can use the `yum update-to` command. `yum update-to` updates a package to an explicit version instead of the latest:

```
yum update-to bash-4.2.46-28.37.amzn1
```

Updates Bash to version 4.2.46-28.37.amznl

Using a CloudFormation template to describe an EC2 instance with explicitly defined updates looks like this:

```
Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      # [...]
      UserData:
        'Fn::Base64': |
          #!/bin/bash -x
          yum update-to bash-4.2.46-28.37.amzn1
```

The same approach works for non-security-related package updates. Whenever a new security update is released, you should check whether you're affected and modify the user data to keep new systems secure.

6.2.3 Installing security updates on running virtual machines

What do you do if you have to install a security update of a core component on tens or even hundreds of virtual machines? You could manually log in to all your VMs using SSH and run `yum -y --security update` or `yum update-to [...]`, but if you have many machines or the number of machines grows, this can be annoying. One way to automate this task is to use a small script that gets a list of your VMs and executes `yum` in all of them. The following listing shows how this can be done in with the help of a Bash script. You can find the code in `/chapter06/update.sh` in the book's code folder.

Listing 6.1 Installing security updates on all running EC2 instances

```
PUBLICIPADDRESSES="$(aws ec2 describe-instances \
➥ --filters "Name=instance-state-name,Values=running" \
➥ --query "Reservations[].Instances[].PublicIpAddress" \
➥ --output text)"

for PUBLICIPADDRESS in $PUBLICIPADDRESSES; do
  ssh -t "ec2-user@$PUBLICIPADDRESS" \
➥ "sudo yum -y --security update"
done
```

Gets all public names of running EC2 instances

Connects via SSH

Executes the yum update command

Now you can quickly apply updates to all of your running machines.

AWS Systems Manager: apply patches in an automated way

Using SSH to install security updates on all your virtual machines is challenging. You need to have a network connection as well as a key for each virtual machine. Handling errors during applying patches is another challenge.

The AWS Systems Manager service is a powerful tool when managing virtual machines. First you install an agent on each virtual machine. You then control your EC2 instances with the help of AWS SSM, for example by creating a job to patch all your EC2 instances with the latest patch level from the AWS Management Console.

Some security updates require you to reboot the VM—for example, if you need to patch the kernel of your VMs running on Linux. You can automate the reboot of the machines or switch to an updated AMI and start new virtual machines instead. For example, new AMIs of Amazon Linux including the latest packages are released frequently.

6.3 Securing your AWS account

Securing your AWS account is critical. If someone gets access to your AWS account, they can steal your data, use resources at your expense, or delete all your data. As figure 6.2 shows, an AWS account is a basket for all the resources you own: EC2 instances, CloudFormation stacks, IAM users, and so on. Each AWS account comes with a root user granted unrestricted access to all resources. So far, you've used the root user to log into the Management Console and the user `mycli`—created in section 4.2—when using the CLI. In this section you will create an additional user to log into the Management Console, to avoid using the root user at all. Doing so allows you to manage multiple users, each restricted to the resources that are necessary for their roles.

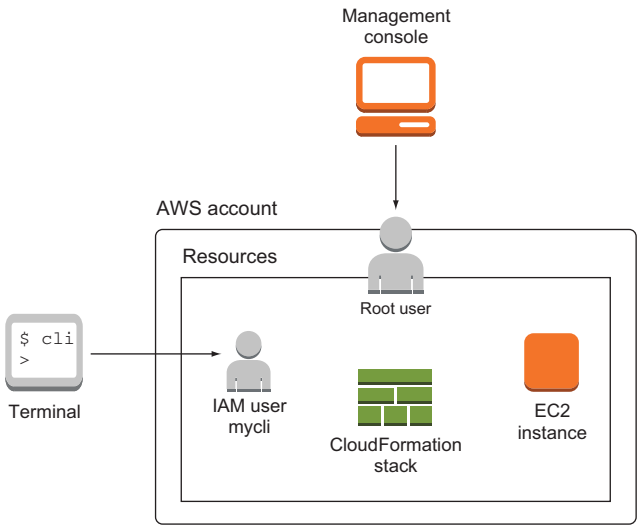


Figure 6.2 An AWS account contains all the AWS resources and comes with a root user by default.

To access your AWS account, an attacker must be able to authenticate to your account. There are three ways to do so: using the root user, using a normal user, or authenticating as an AWS resource like an EC2 instance. To authenticate as a root user or normal user, the attacker needs the username and password or the access keys. To authenticate as an AWS resource like an EC2 instance, the attacker needs to send API/CLI requests from that machine.

To protect yourself from an attacker stealing or cracking your passwords or access keys, you will enable multi-factor authentication (MFA) for your root user, to add an additional layer of security to the authentication process, in the following section.

6.3.1 Securing your AWS account's root user

We advise you to enable MFA for the root user of your AWS account. After MFA is activated, a password and a temporary token are needed to log in as the root user.

Follow these steps to enable MFA, as shown in figure 6.3:

1. Click your name in the navigation bar at the top of the Management Console.
2. Select My Security Credentials.
3. A pop-up should appear. Click Continue to Security Credentials.
4. Install an MFA app on your smartphone, one that supports the TOTP standard (such as Google Authenticator).
5. Expand the Multi-factor authentication (MFA) section.
6. Click Activate MFA.
7. Select a virtual MFA device and proceed with the next step.
8. Follow the instructions in the wizard. Use the MFA app on your smartphone to scan the QR code that is displayed.

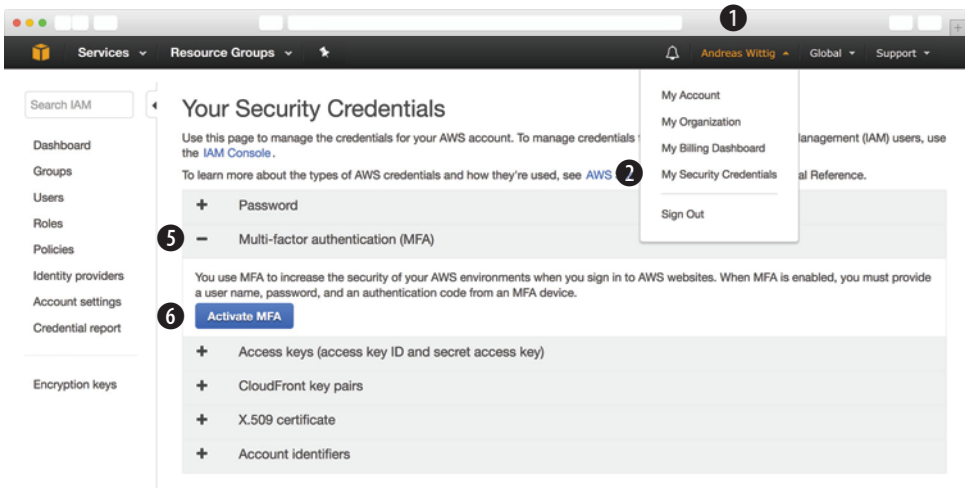


Figure 6.3 Protect your root user with multi-factor authentication (MFA).

If you're using your smartphone as a virtual MFA device, it's a good idea not to log in to the Management Console from your smartphone or to store the root user's password on the phone. Keep the MFA token separate from your password.

6.3.2 AWS Identity and Access Management (IAM)

Figure 6.4 shows an overview of all the core concepts of the Identity and Access Management (IAM) service. This service provides authentication and authorization for the AWS API. When you send a request to the AWS API, IAM verifies your identity and checks if you are allowed to perform the action. IAM controls who (authentication) can do what (authorization) in your AWS account. For example, is the user allowed to launch a new virtual machine?

- An *IAM user* is used to authenticate people accessing your AWS account.
- An *IAM group* is a collection of IAM users.
- An *IAM role* is used to authenticate AWS resources, for example an EC2 instance.
- An *IAM policy* is used to define the permissions for a user, group, or role.

Table 6.1 shows the differences between users and roles. Roles authenticate AWS entities such as EC2 instances. IAM users authenticate the people who manage AWS resources, for example system administrators, DevOps engineers, or software developers.

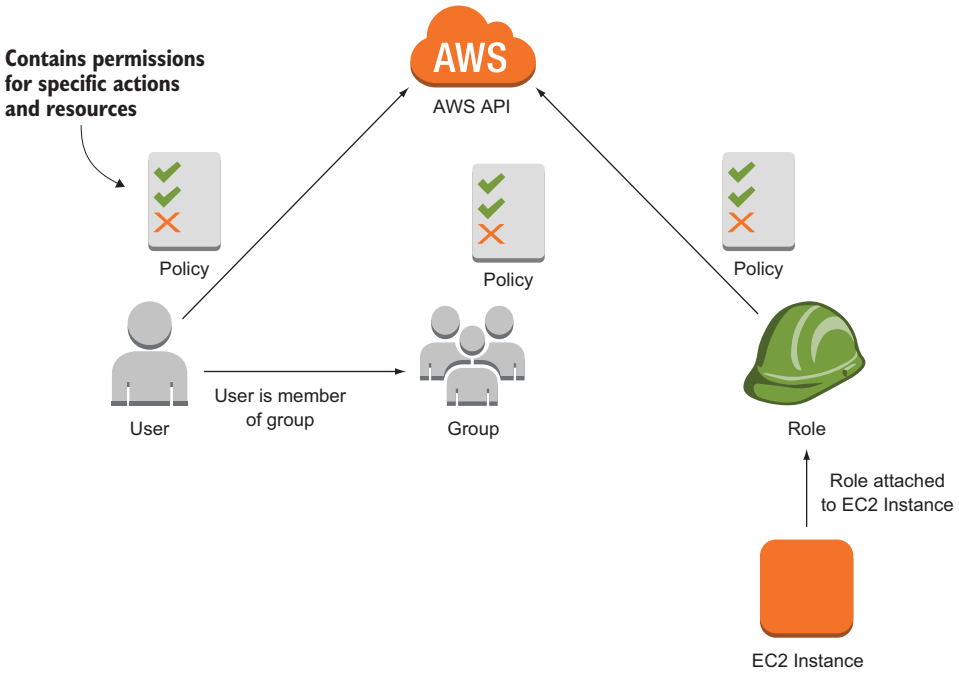


Figure 6.4 IAM concepts

Table 6.1 Differences between root user, IAM user, and IAM role

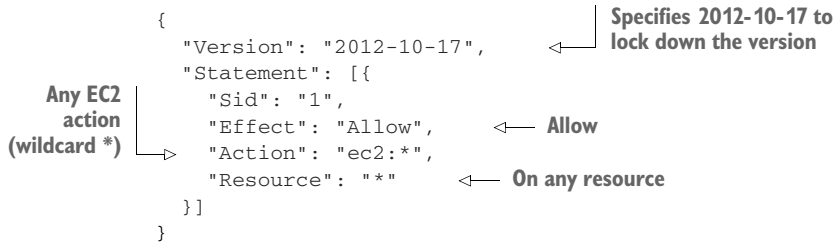
	Root user	IAM user	IAM role
Can have a password (needed to log into the AWS Management Console)	Always	Yes	No
Can have access keys (needed to send requests to the AWS API (for example, for CLI or SDK)	Yes (not recommended)	Yes	No
Can belong to a group	No	Yes	No
Can be associated with an EC2 instance	No	No	Yes

By default, users and roles can't do anything. You have to create a policy stating what actions they're allowed to perform. IAM users and IAM roles use policies for authorization. Let's look at policies first.

6.3.3 Defining permissions with an IAM policy

By attaching one or multiple IAM policies to an IAM user or role, you are granting permissions to manage AWS resources. Policies are defined in JSON and contain one or more statements. A statement can either allow or deny specific actions on specific resources. The wildcard character * can be used to create more generic statements.

The following policy has one statement that allows every action for the EC2 service, for all resources:



If you have multiple statements that apply to the same action, `Deny` overrides `Allow`. The following policy allows all EC2 actions except terminating EC2 instances:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "1",
    "Effect": "Allow",
    "Action": "ec2:*",
    "Resource": "*"
  }, {
    "Sid": "2",
    "Effect": "Deny",
    "Action": "ec2:TerminateInstances",
    "Resource": "*"
  }]
}
```

Deny

Terminating EC2 instances

The following policy denies all EC2 actions. The `ec2:TerminateInstances` statement isn't crucial, because `Deny` overrides `Allow`. When you deny an action, you can't allow that action with another statement:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "1",
    "Effect": "Deny",
    "Action": "ec2:*",
    "Resource": "*"
  }, {
    "Sid": "2",
    "Effect": "Allow",
    "Action": "ec2:TerminateInstances",
    "Resource": "*"
  }]
}
```

Denies every EC2 action

Allow isn't crucial; Deny overrides Allow.

So far, the `Resource` part has been `["*"]` for every resource. Resources in AWS have an Amazon Resource Name (ARN); figure 6.5 shows the ARN of an EC2 instance.

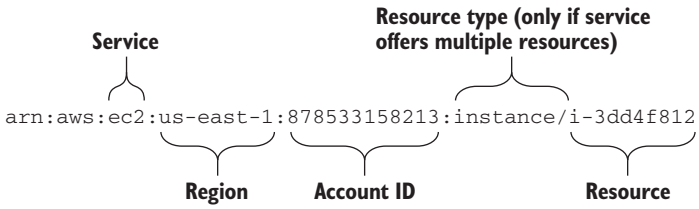


Figure 6.5 Components of an Amazon Resource Name (ARN) identifying an EC2 instance

To find out the account ID, you can use the CLI:

```
$ aws iam get-user --query "User.Arn" --output text
arn:aws:iam::111111111111:user/mycli
```

Account ID has 12 digits. 111111111111 in our example.

If you know your account ID, you can use ARNs to allow access to specific resources of a service:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "2",
    "Effect": "Allow",
    "Action": "ec2:TerminateInstances",
    "Resource":
➥ "arn:aws:ec2:us-east-1:111111111111:instance/i-0b5c991e026104db9"
  }]
}
```

There are two types of policies:

1. *Managed policy*—If you want to create policies that can be reused in your account, a managed policy is what you're looking for. There are two types of managed policies:
   - *AWS managed policy*—A policy maintained by AWS. There are policies that grant admin rights, read-only rights, and so on.
   - *Customer managed*—A policy maintained by you. It could be a policy that represents the roles in your organization, for example.
2. *Inline policy*—A policy that belongs to a certain IAM role, user, or group. An inline policy can't exist without the IAM role, user, or group that it belongs to.

With CloudFormation, it's easy to maintain inline policies; that's why we use inline policies most of the time in this book. One exception is the `mycli` user: this user has the AWS managed policy `AdministratorAccess` attached. We maintain an open-source list of all possible IAM permissions available at https://iam.cloudonaut.io/.

6.3.4 Users for authentication, and groups to organize users

A user can authenticate using either a user name and password, or access keys. When you log in to the Management Console, you're authenticating with your user name and password. When you use the CLI from your computer, you use access keys to authenticate as the `mycli` user.

You're using the root user at the moment to log in to the Management Console. You should create an IAM user instead, for two reasons.

- Creating IAM users allows you to set up a unique user for every person who needs to access your AWS account.
- You can grant access only to the resources each user needs, allowing you to follow the least privilege principle.

To make things easier if you want to add users in the future, you'll first create a group for all users with administrator access. Groups can't be used to authenticate, but they centralize authorization. So, if you want to stop your admin users from terminating EC2 instances, you only need to change the policy for the group instead of changing it for all admin users. A user can be a member of zero, one, or multiple groups.

It's easy to create groups and users with the CLI. Replace *`$Password`* in the following with a secure password:

```
$ aws iam create-group --group-name "admin"
$ aws iam attach-group-policy --group-name "admin" \
➥ --policy-arn "arn:aws:iam::aws:policy/AdministratorAccess"
$ aws iam create-user --user-name "myuser"
$ aws iam add-user-to-group --group-name "admin" --user-name "myuser"
$ aws iam create-login-profile --user-name "myuser" --password "$Password"
```

The user `myuser` is ready to be used. But you must use a different URL to access the Management Console if you aren't using the root user: https://$accountId.signin.aws.amazon.com/console. Replace `$accountId` with the account ID that you extracted earlier with the `aws iam get-user` command.

Enabling MFA for IAM users

We encourage you to enable MFA for all users. If possible, don't use the same MFA device for your root user that you use for everyday users. You can buy hardware MFA devices for $13 from AWS partners like Gemalto. To enable MFA for your users, follow these steps:

- Open the IAM service in the Management Console.
- Choose Users at left.
- Select the `myuser` user.
- Select the Security credentials tab.
- Click the pencil near to Assigned MFA device. The wizard to enable MFA for the IAM user is the same one you used for the root user.

We do recommend enabling MFA for all users, especially for users granted administrator access to all or some services.

WARNING Stop using the root user from now on. Always use `myuser` and the new link to the Management Console.

WARNING You should never copy a user's access keys to an EC2 instance; use IAM roles instead! Don't store security credentials in your source code. And never ever check them into your Git or SVN repository. Try to use IAM roles instead whenever possible.

6.3.5 Authenticating AWS resources with roles

There are various use cases where an EC2 instance needs to access or manage AWS resources. For example, an EC2 instance might need to :

- Back up data to the object store S3.
- Terminate itself after a job has been completed.
- Change the configuration of the private network environment in the cloud.

To be able to access the AWS API, an EC2 instance needs to authenticate itself. You could create an IAM user with access keys and store the access keys on an EC2 instance for authentication. But doing so is a hassle, especially if you want to rotate the access keys regularly.

Instead of using an IAM user for authentication, you should use an IAM role whenever you need to authenticate AWS resources like EC2 instances. When using an IAM role, your access keys are injected into your EC2 instance automatically.

If an IAM role is attached to an EC2 instance, all policies attached to those roles are evaluated to determine whether the request is allowed. By default, no role is attached to an EC2 instance and therefore the EC2 instance is not allowed to make any calls to the AWS API.

The following example will show you how to use an IAM role for an EC2 instance. Do you remember the temporary EC2 instances from chapter 4? What if we forgot to terminate those VMs? A lot of money was wasted because of that. You'll now create an EC2 instance that stops itself automatically. The following snippet shows a one-liner terminating an EC2 instance after 5 minutes. The command `at` is used to execute the `aws ec2 stop-instances` with a 5 minute delay:

```
echo "aws ec2 stop-instances --instance-ids i-0b5c991e026104db9" \
➥ | at now + 5 minutes
```

The EC2 instance needs permission to stop itself. Therefore, you need to attach an IAM role to the EC2 instance. The role contains an inline policy granting access to the `ec2:StopInstances` action. The following code shows how you define an IAM role with the help of CloudFormation:

```
Role:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Principal:
          Service: 'ec2.amazonaws.com'
        Action:
        - 'sts:AssumeRole'
    Policies:
    - PolicyName: ec2
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Sid: Stmt1425388787000
          Effect: Allow
          Action: 'ec2:StopInstances'
          Resource: '*'
          Condition:
            StringEquals:
              'ec2:ResourceTag/aws:cloudformation:stack-id':
➥ !Ref 'AWS::StackId'
```

Allow the EC2 service to assume this role.

Policies begin

Policy definition

Condition can solve the problem: only allow if tagged with the stack ID.

To attach an inline role to an instance, you must first create an instance profile:

```
InstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'
  Properties:
    Roles:
    - !Ref Role
```

The following listing shows how to attach the IAM role to the virtual machine:

```
Instance:
  Type: 'AWS::EC2::Instance'
  Properties:
    # [...]
    IamInstanceProfile: !Ref InstanceProfile
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -x
        INSTANCEID="$(curl -s http://169.254.169.254/\
➥ latest/meta-data/instance-id)"
        echo "aws ec2 stop-instances --instance-ids $INSTANCEID \
➥ --region ${AWS::Region}" | at now + ${Lifetime} minutes
```

Create the CloudFormation stack with the template located at http://mng.bz/Z35X by clicking on the CloudFormation Quick-Create link: http://mng.bz/833J. Specify the lifetime of the EC2 instance via the parameter, and pick the default VPC and subnet as well. Wait until the amount of time specified as the lifetime has passed, and see if your EC2 instance is stopped in the EC2 Management Console. The lifetime begins when the server is fully started and booted.

Cleaning up

Don't forget to delete your stack `ec2-iamrole` after you finish this section, to clean up all used resources. Otherwise you'll likely be charged for the resources you use (even when your EC2 instance is stopped, you pay for the network attached storage).

You have learned how to use IAM users to authenticate people and IAM roles to authenticate EC2 instances or other AWS resources. You've also seen how to grant access to specific actions and resources by using an IAM policy. The next section will cover controlling network traffic to and from your virtual machine.

6.4 Controlling network traffic to and from your virtual machine

You only want traffic to enter or leave your EC2 instance if it has to do so. With a firewall, you control ingoing (also called *inbound* or *ingress*) and outgoing (also called *outbound* or *egress*) traffic. If you run a web server, the only ports you need to open to the outside world are port 80 for HTTP traffic and 443 for HTTPS traffic. All other ports should be closed down. You should only open ports that must be accessible, just as you grant only the permissions you need with IAM. If you are using a firewall that allows only legitimate traffic, you close a lot of possible security holes. You can also prevent yourself from human failure, for example you prevent accidentally sending email to customers from a test system by not opening outgoing SMTP connections for test systems.

Before network traffic can enter or leave your EC2 instance, it goes through a firewall provided by AWS. The firewall inspects the network traffic and uses rules to decide whether the traffic is allowed or denied.

IP vs. IP address

The abbreviation *IP* is used for Internet Protocol, whereas an *IP address* describes a specific address like 84.186.116.47.

Figure 6.6 shows how an SSH request from a source IP address 10.0.0.10 is inspected by the firewall and received by the destination IP address 10.10.0.20. In this case, the firewall allows the request because there is a rule that allows TCP traffic on port 22 between the source and the destination.

Source versus destination

Inbound security-group rules filter traffic based on its source. The source is either an IP address or a security group. Thus you can allow inbound traffic only from specific source IP address ranges.

Outbound security-group rules filter traffic based on its destination. The destination is either an IP address or a security group. You can allow outbound traffic to only specific destination IP address ranges.

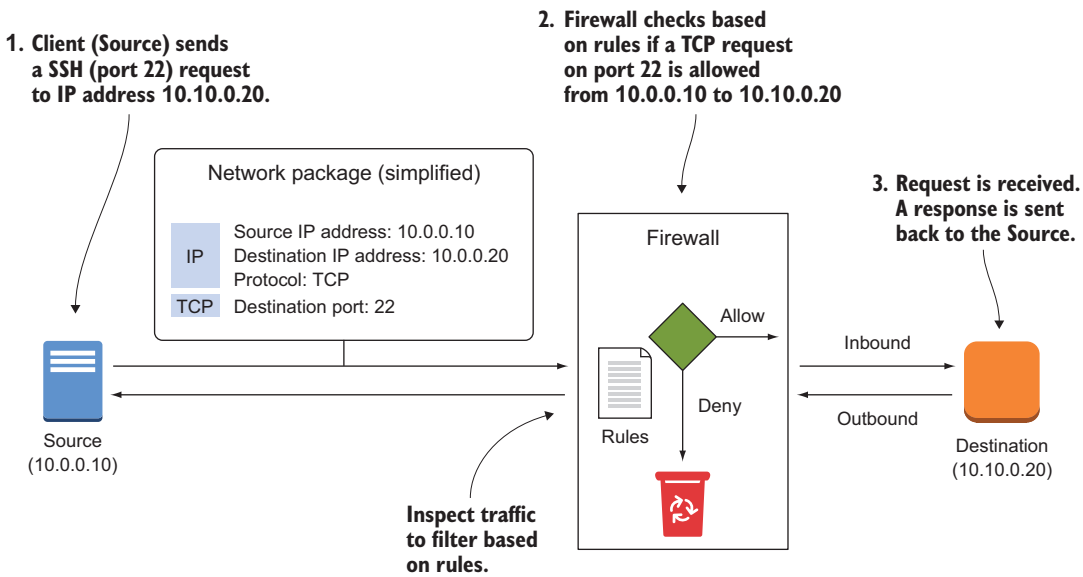


Figure 6.6 How an SSH request travels from source to destination, controlled by a firewall

AWS is responsible for the firewall, but you're responsible for the rules. By default, a security group does not allow any inbound traffic. You must add your own rules to allow specific incoming traffic. A security group contains a rule allowing all outbound traffic by default. If your use case requires a high level of networking security, you should remove the rule and add your own rules to control outgoing traffic.

Debugging or monitoring network traffic

Imagine the following problem: Your EC2 instance does not accept SSH traffic as you want it to, but you can't spot any misconfiguration in your firewall rules. In this case, you should enable VPC Flow Logs to get access to aggregated log messages containing rejected connections. Go to the VPC Flow Logs (AWS Documentation) at http://mng.bz/ltgm to learn more.

VPC Flow Logs give you insight into both granted connections and rejected connections.

6.4.1 Controlling traffic to virtual machines with security groups

Associate a security group with AWS resources such as EC2 instances to control traffic. It's common for EC2 instances to have more than one security group associated with them, and for the same security group to be associated with multiple EC2 instances.

A security group consists of a set of rules. Each rule allows network traffic based on the following:

- Direction (inbound or outbound)
- IP protocol (TCP, UDP, ICMP)
- Port
- Source/destination based on IP address, IP address range, or security group (works only within AWS)

In theory, you could define rules that allow all traffic to enter and leave your virtual machine; AWS won't prevent you from doing so. But it's best practice to define your rules so they are as restrictive as possible.

Security group resources in CloudFormation are of type `AWS::EC2::SecurityGroup`. The following listing is in /chapter06/firewall1.yaml in the book's code folder: the template describes an empty security group associated with a single EC2 instance.

Listing 6.2 CloudFormation template: security group

```
---
[..]
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
  VPC:
```

You'll learn about this in section 6.5.

```
    [..]
  Subnet:
    [..]
Resources:
  SecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Learn how to protect your EC2 Instance.'
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: 'AWS in Action: chapter 6 (firewall)'
  Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: 't2.micro'
      KeyName: !Ref KeyName
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref SecurityGroup
        SubnetId: !Ref Subnet
      Tags:
      - Key: Name
        Value: 'AWS in Action: chapter 6 (firewall)'
```

You'll learn about this in section 6.5.

Defines the security group without any rules (by default, inbound traffic is denied and outbound traffic is allowed.) Rules will be added in the following sections.

Defines the EC2 instance

Associates the security group with the EC2 instance

To explore security groups, you can try the CloudFormation template located at http://mng.bz/fVu5. Create a stack based on that template by clicking on the CloudFormation Quick-Create link (http://mng.bz/Qk5e), and then copy the `PublicName` from the stack output.

6.4.2 Allowing ICMP traffic

If you want to ping an EC2 instance from your computer, you must allow inbound Internet Control Message Protocol (ICMP) traffic. By default, all inbound traffic is blocked. Try `ping` *$PublicName* to make sure `ping` isn't working:

```
$ ping ec2-52-5-109-147.compute-1.amazonaws.com
PING ec2-52-5-109-147.compute-1.amazonaws.com (52.5.109.147): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
[...]
```

You need to add a rule to the security group that allows inbound traffic, where the protocol equals ICMP. Listing 6.3 is in /chapter06/firewall2.yaml in the book's code folder.

Listing 6.3 CloudFormation template: security group that allows ICMP

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Learn how to protect your EC2 Instance.'
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'AWS in Action: chapter 6 (firewall)'
    # allowing inbound ICMP traffic
    SecurityGroupIngress:
    - IpProtocol: icmp
      FromPort: '-1'
      ToPort: '-1'
      CidrIp: '0.0.0.0/0'
```

Rules allowing incoming traffic

Specifies ICMP as the protocol

ICMP does not use ports. -1 means every port.

Allow traffic from any source IP address.

Update the CloudFormation stack with the template located at http://mng.bz/0caa and retry the `ping` command. It should work now:

```
$ ping ec2-52-5-109-147.compute-1.amazonaws.com
PING ec2-52-5-109-147.compute-1.amazonaws.com (52.5.109.147): 56 data bytes
64 bytes from 52.5.109.147: icmp_seq=0 ttl=49 time=112.222 ms
64 bytes from 52.5.109.147: icmp_seq=1 ttl=49 time=121.893 ms
[...]
round-trip min/avg/max/stddev = 112.222/117.058/121.893/4.835 ms
```

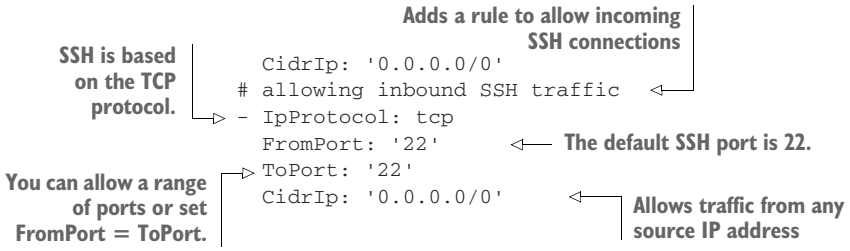
Everyone's inbound ICMP traffic (every source IP address) is now allowed to reach your EC2 instance.

6.4.3 Allowing SSH traffic

Once you can ping your EC2 instance, you want to log in to your virtual machine via SSH. To do so, you must create a rule to allow inbound TCP requests on port 22.

Listing 6.4 CloudFormation template: security group that allows SSH

```
SecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Learn how to protect your EC2 Instance.'
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'AWS in Action: chapter 6 (firewall)'
    # allowing inbound ICMP traffic
    SecurityGroupIngress:
    - IpProtocol: icmp
      FromPort: '-1'
      ToPort: '-1'
```



Update the CloudFormation stack with the template located at http://mng.bz/P8cm. You can now log in to your EC2 instance using SSH. Keep in mind that you still need the correct private key. The firewall only controls the network layer; it doesn't replace key-based or password-based authentication.

6.4.4 Allowing SSH traffic from a source IP address

So far, you're allowing inbound traffic on port 22 (SSH) from every source IP address. It is possible to restrict access to only your own IP address for additional security as well.

> **What's the difference between public and private IP addresses?**
>
> On my local network, I'm using private IP addresses that start with 192.168.0.*. My laptop uses 192.168.0.10, and my iPad uses 192.168.0.20. But if I access the internet, I have the same public IP address (such as 79.241.98.155) for my laptop and iPad. That's because only my internet gateway (the box that connects to the internet) has a public IP address, and all requests are redirected by the gateway. (If you want to know more about this, search for *network address translation*.) Your local network doesn't know about this public IP address. My laptop and iPad only know that the internet gateway is reachable under 192.168.0.1 on the private network.
>
> To find your public IP address, visit http://api.ipify.org. For most of us, our public IP address changes from time to time, usually when you reconnect to the internet (which happens every 24 hours in my case).

Hard-coding the public IP address into the template isn't a good solution because your public IP address changes from time to time. But you already know the solution: parameters. You need to add a parameter that holds your current public IP address, and you need to modify the `Security Group`. You can find the following listing in /chapter06/firewall4.yaml in the book's code folder.

Listing 6.5 Security group allows traffic from source IP

```
Parameters:
  # [...]
  IpForSSH:                          <-- Public IP address parameter
    Description: 'Your public IP address to allow SSH access'
    Type: String
```

```
Resources:
  SecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'Learn how to protect your EC2 Instance.'
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: 'AWS in Action: chapter 6 (firewall)'
      # allowing inbound ICMP traffic
      SecurityGroupIngress:
      - IpProtocol: icmp
        FromPort: '-1'
        ToPort: '-1'
        CidrIp: '0.0.0.0/0'
      # allowing inbound SSH traffic
      - IpProtocol: tcp
        FromPort: '22'
        ToPort: '22'
        CidrIp: !Sub '${IpForSSH}/32'
```

Uses $IpForSSH/32 as a value

Update the CloudFormation stack with the template located at http://mng.bz/S2f9. When asked for parameters, type in your public IP address for *$IPForSSH*. Now only your IP address can open SSH connections to your EC2 instance.

> **Classless Inter-Domain Routing (CIDR)**
>
> You may wonder what `/32` means in listing 6.5. To understand what's going on, you need to switch your brain into binary mode. An IP address is 4 bytes or 32 bits long. The `/32` defines how many bits (32, in this case) should be used to form a range of addresses. If you want to define the exact IP address that is allowed, you must use all 32 bits.
>
> But sometimes it makes sense to define a range of allowed IP addresses. For example, you can use `10.0.0.0/8` to create a range between 10.0.0.0 and 10.255.255.255, `10.0.0.0/16` to create a range between 10.0.0.0 and 10.0.255.255, or `10.0.0.0/24` to create a range between 10.0.0.0 and 10.0.0.255. You aren't required to use the binary boundaries (8, 16, 24, 32), but they're easier for most people to understand. You already used `0.0.0.0/0` to create a range that contains every possible IP address.

Now you can control network traffic that comes from outside a virtual machine or goes outside a virtual machine by filtering based on protocol, port, and source IP address.

6.4.5 Allowing SSH traffic from a source security group

It is possible to control network traffic based on whether the source or destination belongs to a specific security group. For example, you can say that a MySQL database can only be accessed if the traffic comes from your web servers, or that only your proxy servers are allowed to access the web servers. Because of the elastic nature of the cloud, you'll likely deal with a dynamic number of virtual machines, so rules based on

source IP addresses are difficult to maintain. This becomes easy if your rules are based on source security groups.

To explore the power of rules based on a source security group, let's look at the concept of a *bastion host* for SSH access (some people call it a *jump box*). The trick is that only one virtual machine, the bastion host, can be accessed via SSH from the internet (it should be restricted to a specific source IP address). All other virtual machines can only be reached via SSH from the bastion host. This approach has two advantages:

- You have only one entry point into your system, and that entry point does nothing but SSH. The chances of this box being hacked are small.
- If one of your virtual machines that's running a web server, mail server, FTP server, and so on is hacked, the attacker can't jump from that machine to all the other machines.

To implement the concept of a bastion host, you must follow these two rules:

- Allow SSH access to the bastion host from 0.0.0.0/0 or a specific source address.
- Allow SSH access to all other virtual machines only if the traffic source is the bastion host.

Figure 6.7 shows a bastion host with two EC2 instances that are only reachable via SSH from the bastion host.

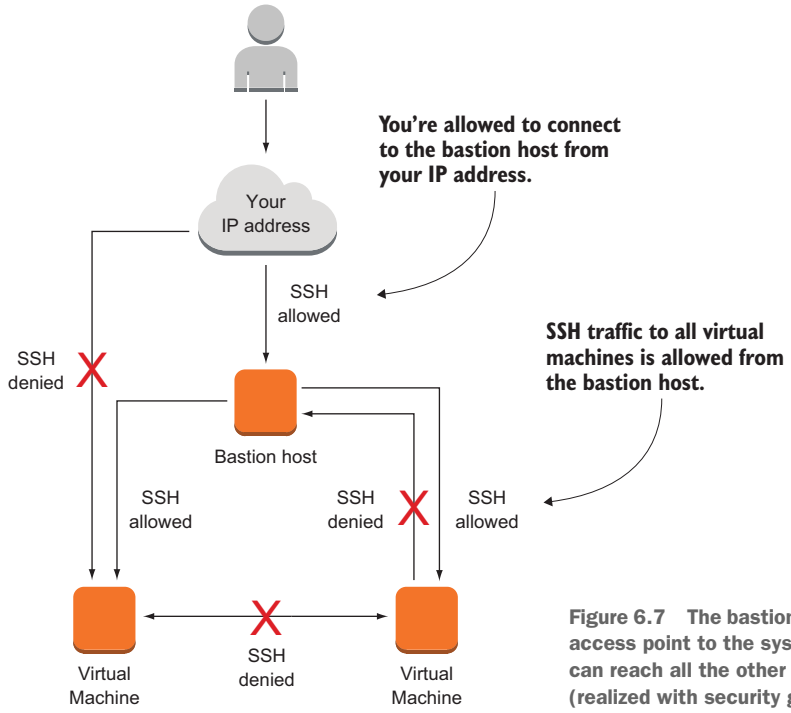


Figure 6.7 The bastion host is the only SSH access point to the system from which you can reach all the other machines via SSH (realized with security groups).

A security group allowing incoming SSH traffic from anywhere needs to be attached to the bastion host. All other VMs are attached to a security group allowing SSH traffic only if the source is the bastion host's security group. The following listing shows the security groups defined in a CloudFormation template:

Listing 6.6 CloudFormation template: SSH from bastion host

```
SecurityGroupBastionHost:                          <-- Security group attached
  Type: 'AWS::EC2::SecurityGroup'                      to the bastion host
  Properties:
    GroupDescription: 'Allowing incoming SSH and ICPM from anywhere.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - IpProtocol: icmp
      FromPort: "-1"
      ToPort: "-1"
      CidrIp: '0.0.0.0/0'
    - IpProtocol: tcp
      FromPort: '22'
      ToPort: '22'                                 <-- Security group
      CidrIp: !Sub '${IpForSSH}/32'                    attached to another
SecurityGroupInstance:                                 virtual machine
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'Allowing incoming SSH from the Bastion Host.'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - IpProtocol: tcp
      FromPort: '22'                                                <-- Allowing incoming
      ToPort: '22'                                                      SSH traffic only
      SourceSecurityGroupId: !Ref SecurityGroupBastionHost              from bastion host
```

Update the CloudFormation stack with the template located at http://mng.bz/VrWk. If the update is completed, the stack will show three outputs:

1. `BastionHostPublicName`—Use the bastion host to connect via SSH from your computer.
2. `Instance1PublicName`—You can connect to this EC2 instance only from the bastion host.
3. `Instance2PublicName`—You can connect to this EC2 instance only from the bastion host.

Execute the following command to add your key to the SSH agent. Replace *`$PathToKey`* with the path to the SSH key:

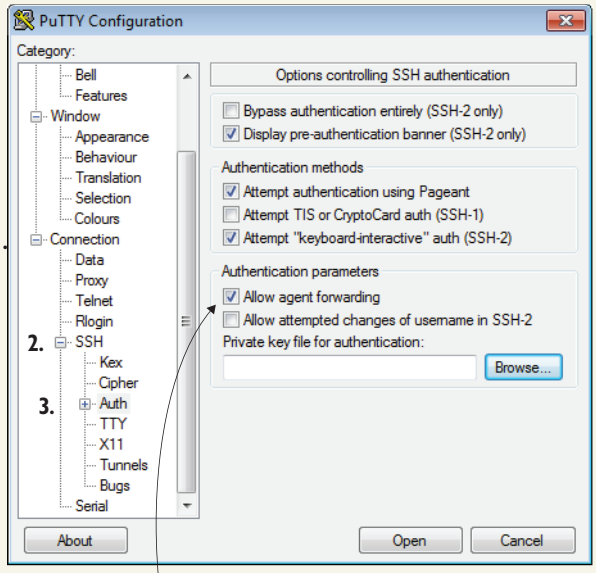
```
ssh-add $PathToKey/mykey.pem
```

Now connect to `BastionHostPublicName` via SSH. Replace *`$BastionHostPublicName`* with the public name of the bastion host.

```
ssh -A ec2-user@$BastionHostPublicName
```

Agent forwarding with PuTTY

To make agent forwarding work with PuTTY, you need to make sure your key is loaded into PuTTY Pageant by double-clicking the private key file. You must also enable Connection > SSH > Auth > Allow Agent Forwarding.



The `-A` option is important for enabling `AgentForwarding`; agent forwarding lets you authenticate with the same key you used to log in to the bastion host for further SSH logins initiated from the bastion host.

Log in to `$Instance1PublicName` or `$Instance2PublicName` from the bastion host next.

```
[computer]$ ssh -A ec2-user@ec2-52-4-234-102.[...].com
Last login: Sat Apr 11 11:28:31 2015 from [...]
[...]
[bastionh]$ ssh ec2-52-4-125-194.compute-1.amazonaws.com
Last login: Sat Apr 11 11:28:43 2015 from [...]
[...]
```

Log in to the bastion host.

Log in to $Instance1PublicName from the bastion host.

The bastion host can be used to add a layer of security to your system. If one of your virtual machines is compromised, the attacker can't jump to other machines in your system. This reduces the potential damage the attacker can inflict. It's important that the bastion host does nothing but SSH, to reduce the chance of it becoming a security problem. We use the bastion-host pattern frequently to protect our clients' infrastructure.

Using agent forwarding is a security risk

We are using agent forwarding in our examples when establishing a SSH connection from the bastion host to one of the other two instances. Agent forwarding is a potential security risk, because the bastion host is able to read the private key from your computer. Therefore, you need to fully trust the bastion host when using agent forwarding.

A more secure alternative is using the bastion host as a proxy. The following command establishes an SSH connection to instance 1 by using the bastion host as a proxy.

```
ssh -J ec2-user@BastionHostPublicName ec2-user@Instance1PublicName
```

In this case the bastion host does not need to access the private key and you can disable agent forwarding.

Cleaning up

Don't forget to delete your stack after you finish this section, to clean up all used resources. Otherwise you'll likely be charged for the resources you use.

6.5 Creating a private network in the cloud: Amazon Virtual Private Cloud (VPC)

When you create a VPC, you get your own private network on AWS. *Private* means you can use the address ranges 10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16 to design a network that isn't necessarily connected to the public internet. You can create subnets, route tables, ACLs, and gateways to the internet or a VPN endpoint .

Subnets allow you to separate concerns. Create a separate subnet for your databases, web servers, proxy servers, or application servers, or whenever you can separate two systems. Another rule of thumb is that you should have at least two subnets: public and private. A public subnet has a route to the internet; a private subnet doesn't. Your load balancer or web servers should be in the public subnet, and your database should reside in the private subnet.

For the purpose of understanding how a VPC works, you'll create a VPC to host an enterprise web application. You'll re-implement the bastion host concept from the previous section by creating a public subnet that contains only the bastion host server. You'll also create a private subnet for your web servers and one public subnet for your proxy servers. The proxy servers absorb most of the traffic by responding with the latest version of the page they have in their cache, and they forward traffic to the private web servers. You can't access a web server directly over the internet—only through the web caches.

The VPC uses the address space 10.0.0.0/16. To separate concerns, you'll create two public subnets and one private subnet in the VPC:

- 10.0.1.0/24 public SSH bastion host subnet
- 10.0.2.0/24 public Varnish proxy subnet
- 10.0.3.0/24 private Apache web server subnet

> **What does 10.0.0.0/16 mean?**
> 10.0.0.0/16 represents all IP addresses in 10.0.0.0 and 10.0.255.255. It's using CIDR notation (explained earlier in the chapter).

Network ACLs restrict traffic that goes from one subnet to another, acting as a firewall. That's an additional layer of security on top of security groups, which control traffic to and from a virtual machine. The SSH bastion host from section 6.4 can be implemented with these ACLs:

- SSH from 0.0.0.0/0 to 10.0.1.0/24 is allowed.
- SSH from 10.0.1.0/24 to 10.0.2.0/24 is allowed.
- SSH from 10.0.1.0/24 to 10.0.3.0/24 is allowed.

To allow traffic to the Varnish proxy and the Apache web servers, you'll need these additional ACLs:

- HTTP from 0.0.0.0/0 to 10.0.2.0/24 is allowed.
- HTTP from 10.0.2.0/24 to 10.0.3.0/24 is allowed.

Figure 6.8 shows the architecture of the VPC.

You'll use CloudFormation to describe the VPC with its subnets. The template is split into smaller parts to make it easier to read in the book. As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. The template is located at /chapter06/vpc.yaml.

6.5.1 Creating the VPC and an internet gateway (IGW)

The first resources listed in the template are the VPC and the internet gateway (IGW). The IGW will translate the public IP addresses of your virtual machines to their private IP addresses using network address translation (NAT). All public IP addresses used in the VPC are controlled by this IGW:

```
VPC:
  Type: 'AWS::EC2::VPC'
  Properties:
    CidrBlock: '10.0.0.0/16'
    EnableDnsHostnames: 'true'
    Tags:
    - Key: Name
      Value: 'AWS in Action: chapter 6 (VPC)'
```

The IP address space used for the private network

Adds a Name tag to the VPC

```
InternetGateway:
  Type: 'AWS::EC2::InternetGateway'
  Properties: {}
VPCGatewayAttachment:
  Type: 'AWS::EC2::VPCGatewayAttachment'
  Properties:
    VpcId: !Ref VPC
    InternetGatewayId: !Ref InternetGateway
```

An IGW is needed to enable traffic to and from the internet.

Attaches the internet gateway to the VPC

Next you'll define the subnet for the bastion host.

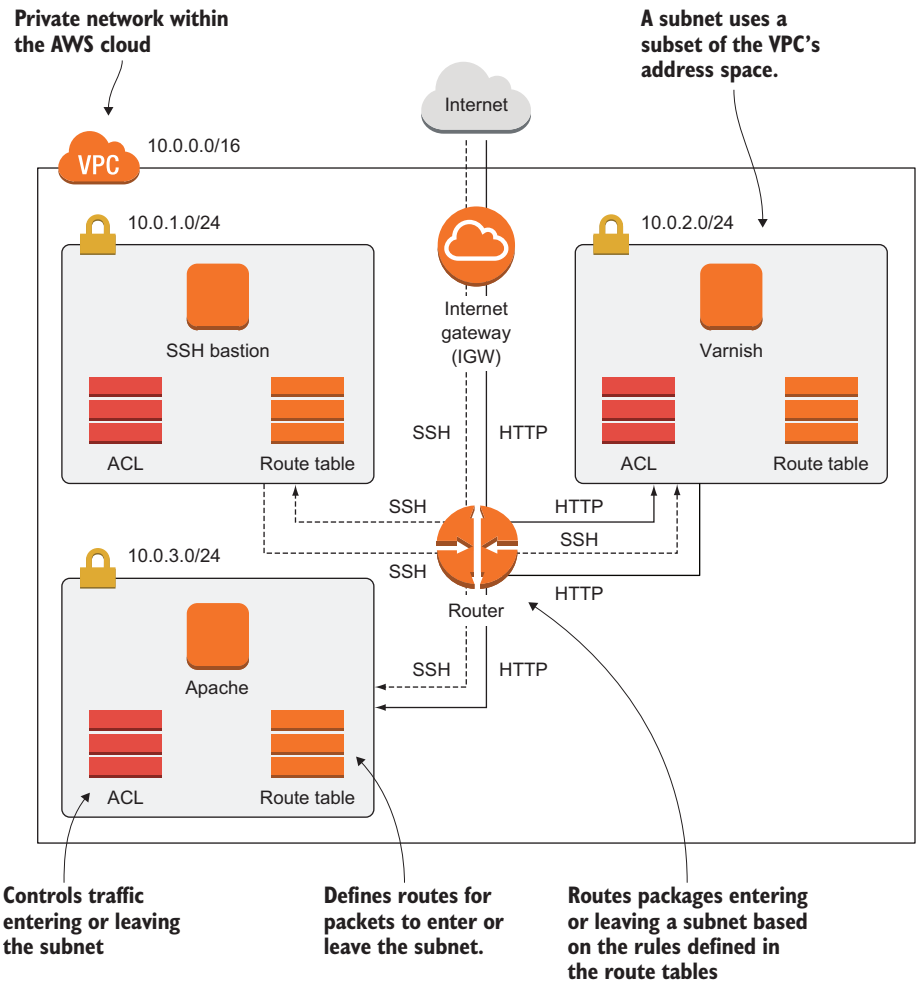


Figure 6.8 VPC with three subnets to secure a web application

6.5.2 Defining the public bastion host subnet

The bastion host subnet will only run a single machine to secure SSH access:

```
SubnetPublicBastionHost:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: 'us-east-1a'
    CidrBlock: '10.0.1.0/24'
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'Public Bastion Host'
RouteTablePublicBastionHost:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPublicBastionHost:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicBastionHost
    RouteTableId: !Ref RouteTablePublicBastionHost
RoutePublicBastionHostToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePublicBastionHost
    DestinationCidrBlock: '0.0.0.0/0'
    GatewayId: !Ref InternetGateway
  DependsOn: VPCGatewayAttachment
NetworkAclPublicBastionHost:
  Type: 'AWS::EC2::NetworkAcl'
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationPublicBastionHost:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetPublicBastionHost
    NetworkAclId: !Ref NetworkAclPublicBastionHost
```

You'll learn about this in chapter 11.

IP address space

Route table

Associates the route table with the subnet

Routes everything (0.0.0.0/0) to the IGW

Network ACL

Associates the NACL with the subnet

The definition of the ACL follows:

```
NetworkAclEntryInPublicBastionHostSSH:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicBastionHost
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '22'
      To: '22'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryInPublicBastionHostEphemeralPorts:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicBastionHost
```

Allows inbound SSH from everywhere

Use the rule number to define the order of rules.

Inbound

Ephemeral ports used for short-lived TCP/IP connections

```
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'false'
    CidrBlock: '10.0.0.0/16'
NetworkAclEntryOutPublicBastionHostSSH:       <-- Allows outbound SSH to VPC
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicBastionHost
    RuleNumber: '100'
    Protocol: '6'
    PortRange:
      From: '22'
      To: '22'
    RuleAction: 'allow'
    Egress: 'true'              <-- Outbound
    CidrBlock: '10.0.0.0/16'
NetworkAclEntryOutPublicBastionHostEphemeralPorts: <-- Ephemeral ports
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAclPublicBastionHost
    RuleNumber: '200'
    Protocol: '6'
    PortRange:
      From: '1024'
      To: '65535'
    RuleAction: 'allow'
    Egress: 'true'
    CidrBlock: '0.0.0.0/0'
```

There's an important difference between security groups and ACLs: security groups are stateful, but ACLs aren't. If you allow an inbound port on a security group, the corresponding response to requests on that port are allowed as well. A security group rule will work as you expect it to. If you open inbound port 22 on a security group, you can connect via SSH.

That's not true for ACLs. If you open inbound port 22 on an ACL for your subnet, you still may not be able to connect via SSH. In addition, you need to allow outbound ephemeral ports, because sshd (SSH daemon) accepts connections on port 22 but uses an ephemeral port for communication with the client. Ephemeral ports are selected from the range starting at 1024 and ending at 65535.

If you want to make a SSH connection from within your subnet, you have to open outbound port 22 and inbound ephemeral ports as well.

There is another difference between security group rules and ACL rules: you have to define the priority for ACL rules. A smaller rule number indicates a higher priority. When evaluating an ACL the first rule that matches a package is applied; all other rules are skipped.

We do recommend to start with using security groups to control traffic. If you want to add an extra layer of security, you should use ACL on top.

6.5.3 Adding the private Apache web server subnet

The subnet for the Varnish web cache is similar to the bastion host subnet because it's also a public subnet; that's why we'll skip it. You'll continue with the private subnet for the Apache web server:

```
SubnetPrivateApacheWebserver:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: 'us-east-1a'
    CidrBlock: '10.0.3.0/24'          <-- Address space
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'Private Apache Webserver'
RouteTablePrivateApacheWebserver:     <-- No route to the IGW
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
RouteTableAssociationPrivateApacheWebserver:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetPrivateApacheWebserver
    RouteTableId: !Ref RouteTablePrivateApacheWebserver
```

As shown in figure 6.9, the only difference between a public and a private subnet is that a private subnet doesn't have a route to the IGW.

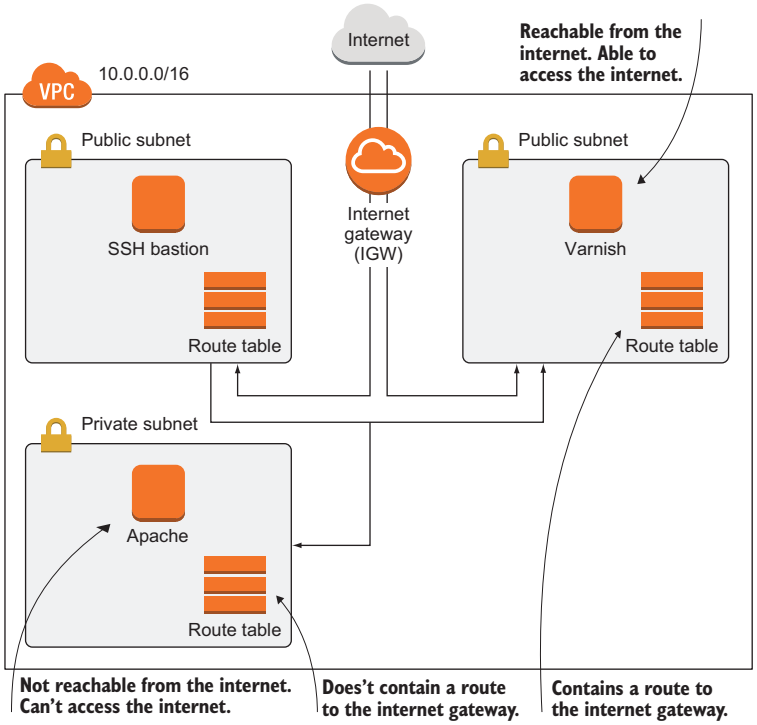


Figure 6.9 Private and public subnets

Traffic between subnets of a VPC is always routed by default. You can't remove the routes between the subnets. If you want to prevent traffic between subnets in a VPC, you need to use ACLs attached to the subnets.

6.5.4 Launching virtual machines in the subnets

Your subnets are ready, and you can continue with the EC2 instances. First you describe the bastion host:

```
BastionHost:
  Type: AWS::EC2::Instance
  Properties:
    ImageId: 'ami-6057e21a'
    InstanceType: 't2.micro'
    KeyName: mykey
    NetworkInterfaces:
    - AssociatePublicIpAddress: true
      DeleteOnTermination: true
      DeviceIndex: '0'
      GroupSet:
      - !Ref SecurityGroup
      SubnetId: !Ref SubnetPublicBastionHost
    Tags:
    - Key: Name
      Value: 'Bastion Host'
  DependsOn: VPCGatewayAttachment
```

Assigns a public IP address

This security group allows everything.

Launches in the bastion host subnet

The Varnish proxy server looks similar. But again, the private Apache web server has a different configuration:

```
ApacheWebserver:
  Type: 'AWS::EC2::Instance'
  Properties:
    ImageId: 'ami-6057e21a'
    InstanceType: 't2.micro'
    KeyName: mykey
    NetworkInterfaces:
    - AssociatePublicIpAddress: false
      DeleteOnTermination: true
      DeviceIndex: '0'
      GroupSet:
      - !Ref SecurityGroup
      SubnetId: !Ref SubnetPrivateApacheWebserver
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -x
        bash -ex << "TRY"
          yum -y install httpd
          service httpd start
        TRY
        /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
    ➥ --resource ApacheWebserver --region ${AWS::Region}
    Tags:
```

No public IP address: private

Launches in the Apache web server subnet

If one of the following two commands fails, the sub-bash stops at this command and returns a non zero exit code.

Installs Apache from the internet

Starts Apache web server

```
    - Key: Name
      Value: 'Apache Webserver'
  CreationPolicy:
    ResourceSignal:
      Timeout: PT10M
  DependsOn: RoutePrivateApacheWebserverToInternet
```

You're now in serious trouble: installing Apache won't work because your private subnet has no route to the internet.

6.5.5 Accessing the internet from private subnets via a NAT gateway

Public subnets have a route to the internet gateway. You can use a similar mechanism to provide internet access for private subnets without having a direct route to the internet: use a NAT gateway in a public subnet, and create a route from your private subnet to the NAT gateway. This way, you can reach the internet from private subnets, but the internet can't reach your private subnets. A NAT gateway is a managed service provided by AWS that handles network address translation. Internet traffic from your private subnet will access the internet from the public IP address of the NAT gateway.

> **Reducing costs for NAT gateway**
>
> You have to pay for the traffic processed by a NAT gateway (see VPC Pricing at https://aws.amazon.com/vpc/pricing/ for more details). If your EC2 instances in private subnets will have to transfer huge amounts of data to the Internet, there are two options to decrease costs.
>
> - Moving your EC2 instances from the private subnet to a public subnet allows them to transfer data to the internet without utilizing the NAT gateway. Use firewalls to strictly restrict incoming traffic from the internet.
> - If data is transferred over the internet to reach AWS services (such as Amazon S3 and Amazon DynamoDB), use so-called VPC endpoints. These endpoints allow your EC2 instances to communicate with S3 and DynamoDB directly without using the NAT gateway. Furthermore, some services are accessible from private subnets via AWS PrivateLink (such as Amazon Kinesis, AWS SSM, and more). Note: AWS PrivateLink is not yet available in all regions.

To keep concerns separated, you'll create a subnet for the NAT gateway.

```
SubnetPublicNAT:
  Type: 'AWS::EC2::Subnet'
  Properties:
    AvailabilityZone: 'us-east-1a'
    CidrBlock: '10.0.0.0/24'
    VpcId: !Ref VPC
    Tags:
    - Key: Name
      Value: 'Public NAT'
```

10.0.0.0/24 is the NAT subnet.

```
RouteTablePublicNAT:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
# [...]
RoutePublicNATToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePublicNAT
    DestinationCidrBlock: '0.0.0.0/0'
    GatewayId: !Ref InternetGateway
  DependsOn: VPCGatewayAttachment
# [...]
EIPNatGateway:
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: 'vpc'
NatGateway:
  Type: 'AWS::EC2::NatGateway'
  Properties:
    AllocationId: !GetAtt 'EIPNatGateway.AllocationId'
    SubnetId: !Ref SubnetPublicNAT
# [...]
RoutePrivateApacheWebserverToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTablePrivateApacheWebserver
    DestinationCidrBlock: '0.0.0.0/0'
    NatGatewayId: !Ref NatGateway
```

The NAT subnet is public with a route to the internet.

A static public IP address is used for the NAT gateway.

The NAT gateway is placed into the private subnet and associated with the static public IP address.

Route from the Apache subnet to the NAT gateway

WARNING The NAT gateway included in the example is not covered by the Free Tier. The NAT gateway will cost you $0.045 USD per hour and $0.045 per GB of data processed when creating the stack in the US East (N. Virginia) region. Go to https://aws.amazon.com/vpc/pricing/ to have a look at the current prices.

Now you're ready to create the CloudFormation stack with the template located at http://mng.bz/NRLj by clicking on the CloudFormation Quick-Create Link: http://mng.bz/71o2. Once you've done so, copy the `VarnishProxyPublicName` output and open it in your browser. You'll see an Apache test page that was cached by Varnish.

Cleaning up

Don't forget to delete your stack after finishing this section, to clean up all used resources. Otherwise you'll likely be charged for the resources you use.

Summary

- AWS is a shared-responsibility environment in which security can be achieved only if you and AWS work together. You're responsible for securely configuring your AWS resources and your software running on EC2 instances, while AWS protects buildings and host systems.
- Keeping your software up-to-date is key, and can be automated.
- The Identity and Access Management (IAM) service provides everything needed for authentication and authorization with the AWS API. Every request you make to the AWS API goes through IAM to check whether the request is allowed. IAM controls who can do what in your AWS account. To protect your AWS account, grant only those permissions that your users and roles need.
- Traffic to or from AWS resources like EC2 instances can be filtered based on protocol, port, and source or destination.
- A bastion host is a well-defined single point of access to your system. It can be used to secure SSH access to your virtual machines. Implementation can be done with security groups or ACLs.
- A VPC is a private network in AWS where you have full control. With VPCs, you can control routing, subnets, ACLs, and gateways to the internet or your company network via VPN. A NAT gateway enables access to the internet from private subnets.
- You should separate concerns in your network to reduce potential damage if, for example, one of your subnets is hacked. Keep every system in a private subnet that doesn't need to be accessed from the public internet, to reduce your attackable surface.

7 Automating operational tasks with Lambda

This chapter covers

- Creating a Lambda function to perform periodic health checks
- Triggering a Lambda function with CloudWatch events
- Searching through your Lambda function's logs with CloudWatch
- Monitoring Lambda functions with CloudWatch alarms
- Configuring IAM roles so Lambda functions can access other services
- Web application, data processing, and IoT with AWS Lambda
- Limitations of AWS Lambda

This chapter is about adding a new tool to your toolbox. The tool we're talking about, AWS Lambda, is as flexible as a Swiss Army Knife. You don't need a virtual machine to run your own code anymore, as AWS Lambda offers execution environments for Java, Node.js, C#, Python, and Go. All you have to do is to implement a

function, upload your code, and configure the execution environment. Afterward, your code is executed within a fully managed computing environment. AWS Lambda is well-integrated with all parts of AWS, enabling you to easily automate operations tasks within your infrastructure. We use AWS to automate our infrastructure regularly. For example, we use it to add and remove instances to a container cluster based on a custom algorithm, and to process and analyze log files.

AWS Lambda offers a maintenance-free and highly available computing environment. You no longer need to install security updates, replace failed virtual machines, or manage remote access (such as SSH or RDP) for administrators. On top of that, AWS Lambda is billed by invocation. Therefore, you don't have to pay for idling resources that are waiting for work (for example, for a task triggered once a day).

In our first example, you will create a Lambda function that performs periodic health checks for your website. This will teach you how to use the Management Console and offer a blueprint for getting started with AWS Lambda quickly. In our second example, you will learn how to write your own Python code and deploy a Lambda function in an automated way using CloudFormation. Your Lambda function will automatically add a tag to newly launched EC2 instances. At the end of the chapter, we'll show you additional use cases like building web applications, Internet of Things (IoT) back ends, or processing data with AWS Lambda.

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

But what is AWS Lambda? Before diving into our first real-world example, we will start with a short introduction.

7.1 Executing your code with AWS Lambda

Computing capacity is available at different layers of abstraction on AWS: virtual machines, containers, and functions. You learned about the virtual machines offered by Amazon's EC2 service in chapter 3. Containers offer another layer of abstraction on top of virtual machines. We don't cover containers as it would go beyond the scope of our book. AWS Lambda provides computing power as well but in a fine-grained manner: an execution environment for small functions, rather than a full-blown operating system or container.

7.1.1 What is serverless?

When reading about AWS Lambda, you might have stumbled upon the term *serverless* The following quote summarizes the confusion created by a catchy and provocative phrase:

> *[...] the word serverless is a bit of a misnomer. Whether you use a compute service such as AWS Lambda to execute your code, or interact with an API, there are still servers running in the background. The difference is that these servers are hidden from you. There's no infrastructure for you to think about and no way to tweak the underlying operating system. Someone else takes care of the nitty-gritty details of infrastructure management, freeing your time for other things.*
>
> —Peter Sbarski, *Serverless Architectures on AWS* (Manning, 2017)

We define a serverless system as one that meets the following criteria:

- No need to manage and maintain virtual machines.
- Fully managed service offering scalability and high availability.
- Billed per request and by resource consumption.
- Invoke the function to execute your code in the cloud.

AWS is not the only provider offering a serverless platform. Google (Cloud Functions) and Microsoft (Azure Functions) are other competitors in this area.

7.1.2 Running your code on AWS Lambda

As shown in figure 7.1, to execute your code with AWS Lambda, follow these steps:

1. Write the code.
2. Upload your code and its dependencies (such as libraries or modules).
3. Create a function determining the runtime environment and configuration.
4. Invoke the function to execute your code in the cloud.

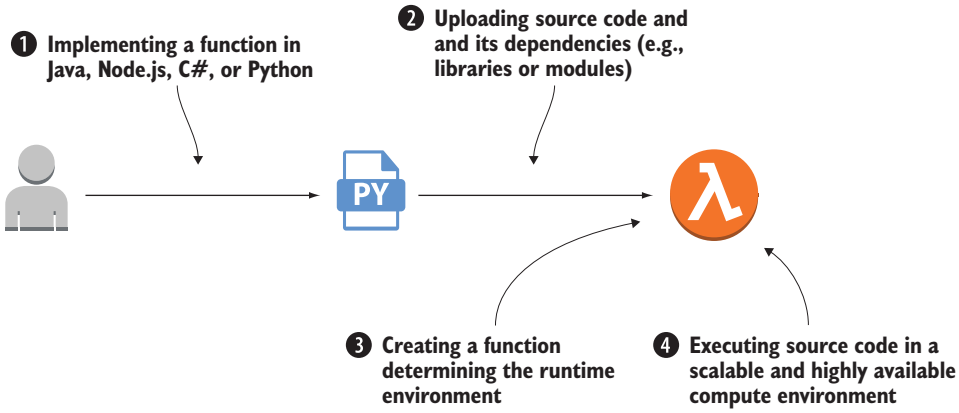


Figure 7.1 Executing code with AWS Lambda

You don't have to launch any virtual machines. AWS executes your code in a fully managed computing environment.

Currently, AWS Lambda offers runtime environments for the following languages:

- Java
- Node.js
- C#
- Python
- Go

Next, we will compare AWS Lambda with EC2 virtual machines.

7.1.3 Comparing AWS Lambda with virtual machines (Amazon EC2)

What is the difference between using AWS Lambda and virtual machines? First, there is the granularity of virtualization. Virtual machines provide a full operating system for running one or multiple applications. In contrast, AWS Lambda offers an execution environment for a single function, a small part of an application.

Furthermore, Amazon EC2 offers VMs as a service, but you are responsible for operating them in a secure, scalable and highly available way. Doing so requires you to put a substantial amount of effort into maintenance. By contrast, AWS Lambda offers a fully managed execution environment. AWS manages the underlying infrastructure for you and provides a production-ready infrastructure.

Beyond that, AWS Lambda is billed per execution, and not per second a virtual machine is running. You don't have to pay for unused resources that are waiting for requests or tasks. For example, running a script to check the health of a website every 5 minutes on a virtual machine would cost you a minimum of $4 USD. Executing the same health check with AWS Lambda is free: you don't even exceed the monthly Free Tier of AWS Lambda.

Table 7.1 compares AWS Lambda and virtual machines in detail. You'll find a discussion of AWS Lambda's limitations at the end of the chapter.

Table 7.1 AWS Lambda compared to Amazon EC2

	AWS Lambda	Amazon EC2
Granularity of virtualization	Small piece of code (a function)	A whole operating system
Scalability	Scales automatically. A throttling limit prevents you from creating unwanted charges accidentally and can be increased by AWS support if needed.	As discussed in chapter 17, using an Auto Scaling Group allows you to scale the number of EC2 instances serving requests automatically. But configuring and monitoring the scaling activities is your responsibility.
High availability	Fault tolerant by default. The computing infrastructure spans multiple machines and data centers.	VMs are not highly available by default. Nevertheless, as you will learn in chapter 14 it is possible to set up a highly available infrastructure based on EC2 instances as well.

Table 7.1 AWS Lambda compared to Amazon EC2 *(continued)*

	AWS Lambda	Amazon EC2
Maintenance effort	Almost zero. You need only to configure your function.	You are responsible for maintaining all layers between your virtual machine's operating system and your application's runtime environment.
Deployment effort	Almost zero due to a well-defined API	Rolling out your application to a fleet of VMs is a challenge that requires tools and know-how.
Pricing model	Pay per request as well as execution time and memory	Pay for operating hours of the virtual machines, billed per second.

Looking for limitations and pitfalls of AWS Lambda? Stay tuned: you will find a discussion of Lambda's limitations at the end of the chapter.

That's all you need to know about AWS Lambda to be able to go through the first real-world example. Are you ready?

7.2 Building a website health check with AWS Lambda

Are you responsible for the uptime of a website or application? We do our best to make sure our blog cloudonaut.io is accessible 24/7. An external health check acts as a safety net making sure we, and not our readers, are the first to know when our blog goes down. AWS Lambda is the perfect choice for building a website health check, as you do not need computing resources constantly, but only every few minutes for a few milliseconds. This section guides you through setting up a health check for your website based on AWS Lambda.

In addition to AWS Lambda, we are using the Amazon CloudWatch service for this example. Lambda functions publish metrics to CloudWatch by default. Typically you inspect metrics using charts, and create alarms by defining thresholds. For example, a metric could count failures during the function's execution. On top of that, CloudWatch provides events that can be used to trigger Lambda functions as well. We are using a schedule to publish an event every 5 minutes here.

As shown in figure 7.2, your website health check will consist of three parts:

1. *Lambda function*—Executes a Python script that sends an HTTP request to your website (for example `GET https://cloudonaut.io`) and verifies that the response includes specific text (such as `cloudonaut`).
2. *Scheduled event*—Triggers the Lambda function every 5 minutes. This is comparable to the cron service on Linux.
3. *Alarm*—Monitors the number of failed health checks and notifies you via email whenever your website is unavailable.

You will use the Management Console to create and configure all the necessary parts manually. In our opinion this is a simple way to get familiar with AWS Lambda. You will learn how to deploy a Lambda function in an automated way in section 7.3.

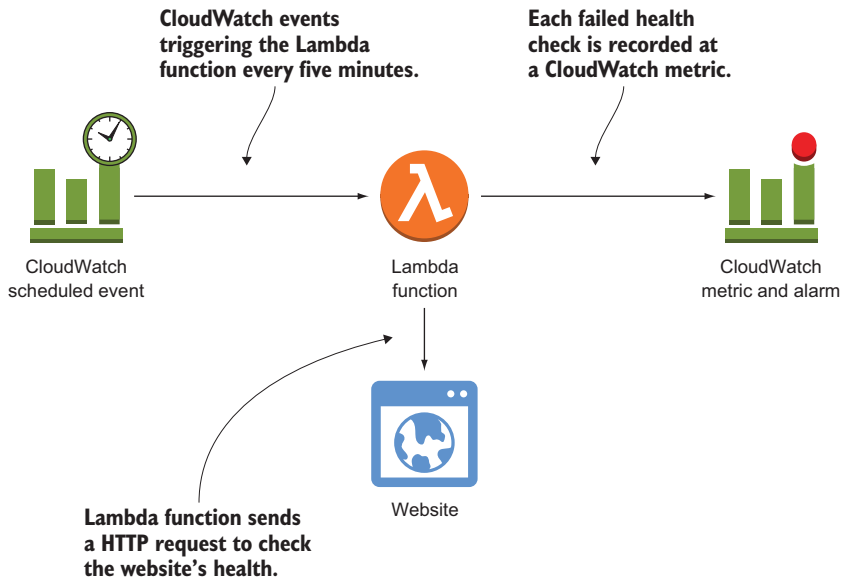


Figure 7.2 The Lambda function performing the website health check is executed every five minutes by a scheduled event. Errors are reported to CloudWatch.

7.2.1 *Creating a Lambda function*

The following step-by-step instructions guide you through setting up a website health check based on AWS Lambda. Open AWS Lambda in the Management Console: https://console.aws.amazon.com/lambda/home. Click Create a function to start the Lambda function wizard, as shown in figure 7.3.

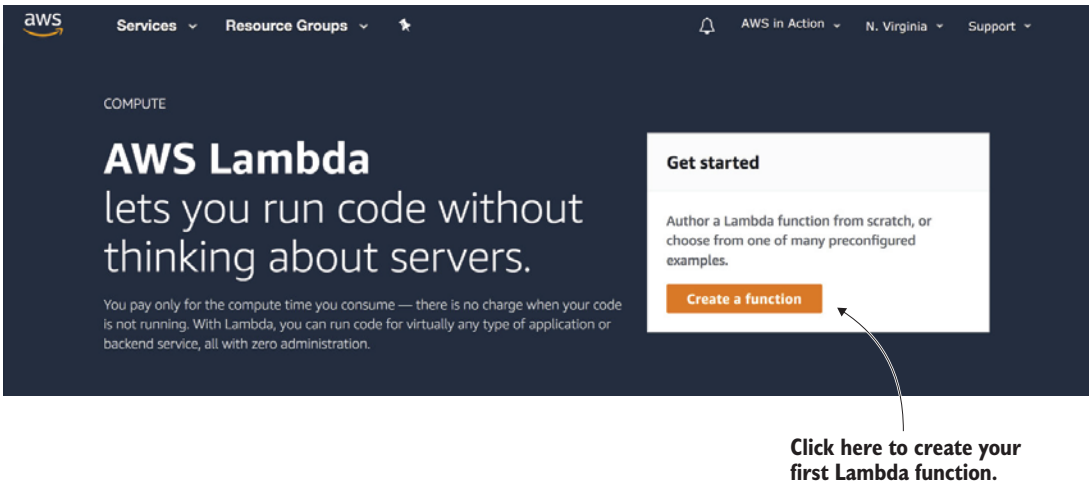


Figure 7.3 Welcome screen: ready to create your first Lambda function

AWS provides blueprints for various use cases, including the code and the Lambda function configuration. We will use one of these blueprints to create a website health check. Select Blueprints and search for `canary`. Next, click the heading of the `lambda-canary-python3` blueprint. Figure 7.4 illustrates the details.

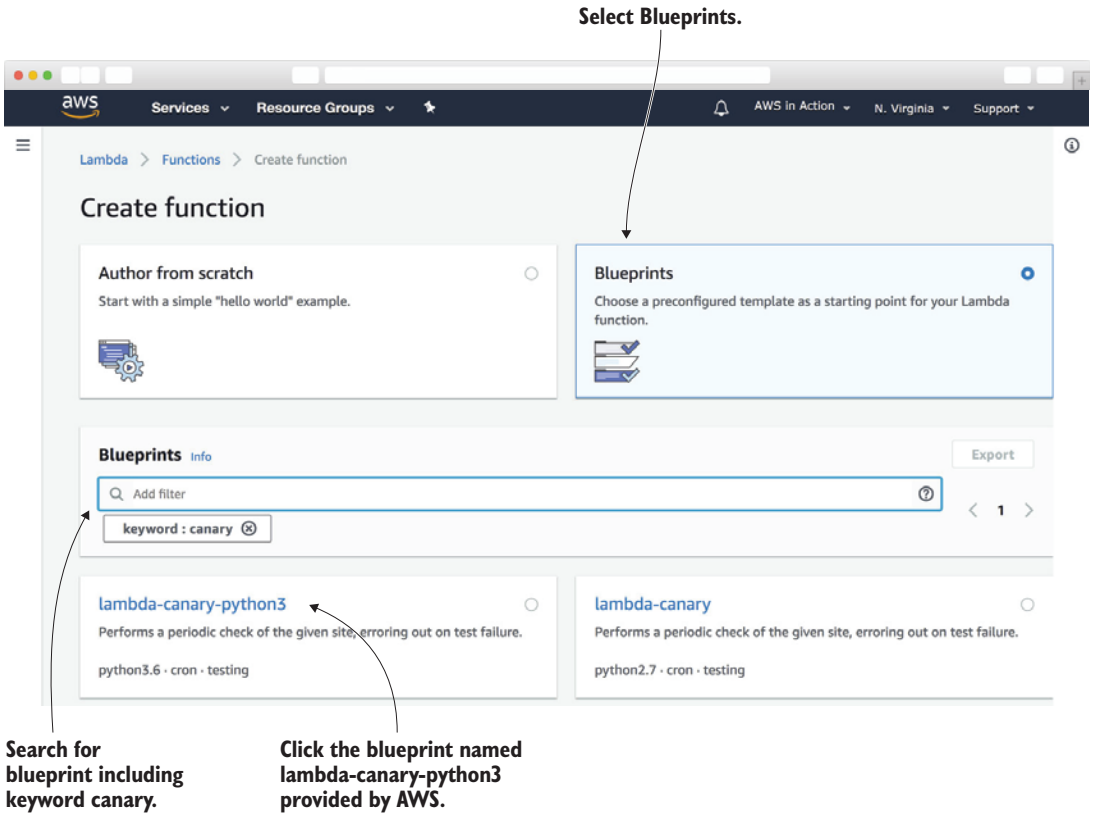


Figure 7.4 Creating a Lambda function based on a blueprint provided by AWS

In the next step of the wizard, you need to specify a name for your Lambda function, as shown in figure 7.5. The function name needs to be unique within your AWS account as well as within the current region US East (N. Virginia), and is limited to 64 characters. To invoke a function via the API you need to provide the function name, for example. Type in `website-health-check` as the name for your Lambda function.

Select Create a custom role as shown in figure 7.5 to create an IAM role for your Lambda function. You will learn how your Lambda function makes use of the IAM role in section 7.3.

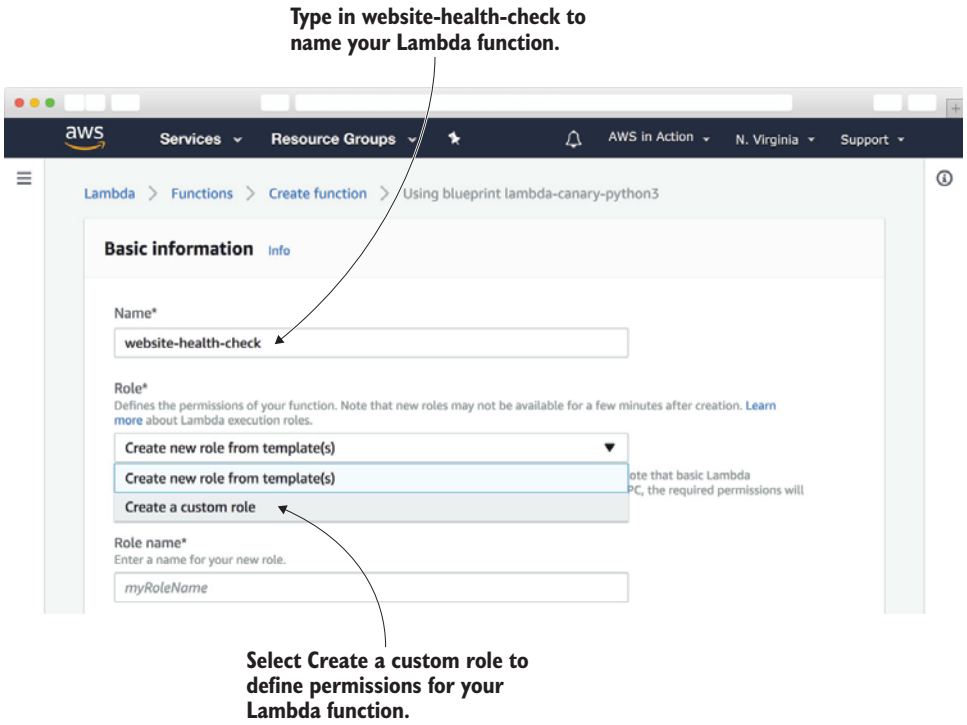


Figure 7.5 Creating a Lambda function: choose a name and define an IAM role

Figure 7.6 illustrates the steps to create a basic IAM role granting your Lambda function write access to CloudWatch logs:

1. Select Create a new IAM role.
2. Keep the role name `lambda_basic_execution`.
3. Click the Allow button.
4. Select the role `lambda_basic_execution` from the drop-down list of existing roles.

You have now specified a name and an IAM role for your Lambda function. Next, you will configure the scheduled event that will trigger your health check repeatedly. We will use an interval of 5 minutes in this example. Figure 7.7 shows the settings you need.

1. Select Create a new rule to create a scheduled event rule.
2. Type in `website-health-check` as the name for the rule.
3. Enter a description that will help you to understand what is going on if you come back later.
4. Select Schedule expression as the rule type. You will learn about the other option, Event pattern, at the end of this chapter.
5. Use `rate(5 minutes)` as the schedule expression.
6. Don't forget to enable the trigger by checking the box at the bottom.

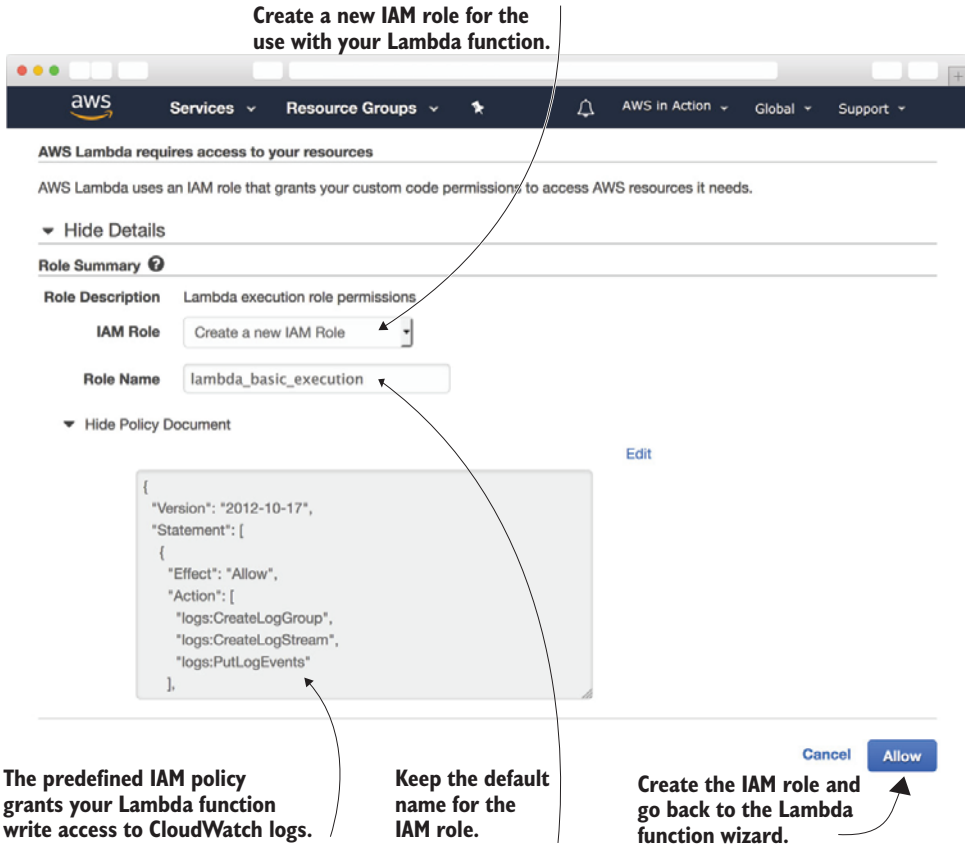


Figure 7.6 Creating an IAM role allowing write access to CloudWatch logs for your Lambda function

You define recurring tasks without the need for a specific time using a *schedule expression* in form of `rate(`*$value $unit*`)`. For example, you could trigger a task every 5 minutes, every hour, or once a day. *$value* needs to be a positive integer value. Use `minute`, `minutes`, `hour`, `hours`, `day`, or `days` as the unit. For example, instead of triggering the website health check every 5 minutes, you could use `rate(1 hour)` as the schedule expression to execute the health check every hour. Note that frequencies of less than one minute are not supported.

It is also possible to use the crontab format when defining a schedule expression.

```
cron($minutes $hours $dayOfMonth $month $dayOfWeek $year)

# Invoke a Lambda function at 08:00am (UTC) everyday
cron(0 8 * * ? *)

# Invoke a Lambda function at 04:00pm (UTC) every monday to friday
cron(0 16 ? * MON-FRI *)
```

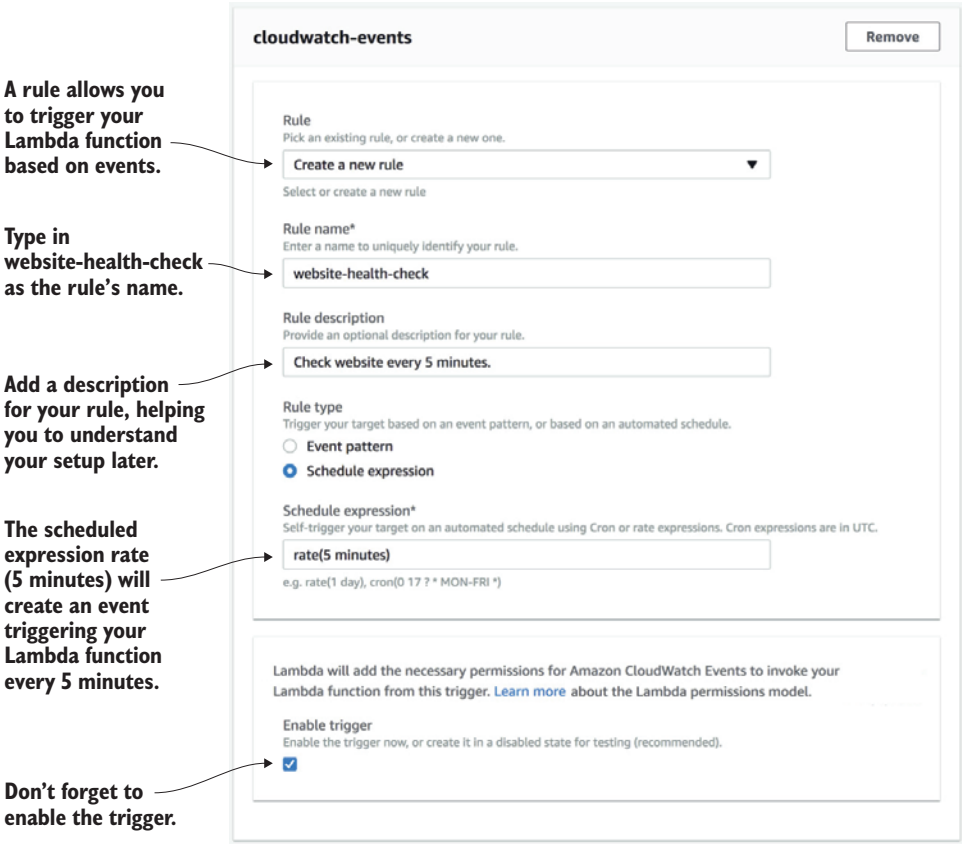


Figure 7.7 Configuring a scheduled event triggering your Lambda function every 5 minutes

See "Schedule Expressions Using Rate or Cron" at http://mng.bz/o35b for more details.

Your Lambda function is missing an integral part: the code. As you are using a blueprint, AWS has inserted the Python code implementing the website health check for you, as shown in figure 7.8.

The Python code references two environment variables: `site` and `expected`. Environment variables are commonly used to dynamically pass settings to your function.

An environment variable consists of a key and a value. Specify the following environment variables for your Lambda function:

1. `site`—Contains the URL of the website you want to monitor. Use https://cloudonaut.io if you do not have a website to monitor yourself.
2. `expected`—Contains a text snippet that must be available on your website. If the function doesn't find this text, the health check fails. Use `cloudonaut` if you are using https://cloudonaut.io as site.

The Lambda function is reading the environment variables during its execution:

```
SITE = os.environment['site']
EXPECTED = os.environment['expected']
```

After defining the environment variables for your Lambda function, click the Create function button at the bottom of the screen.

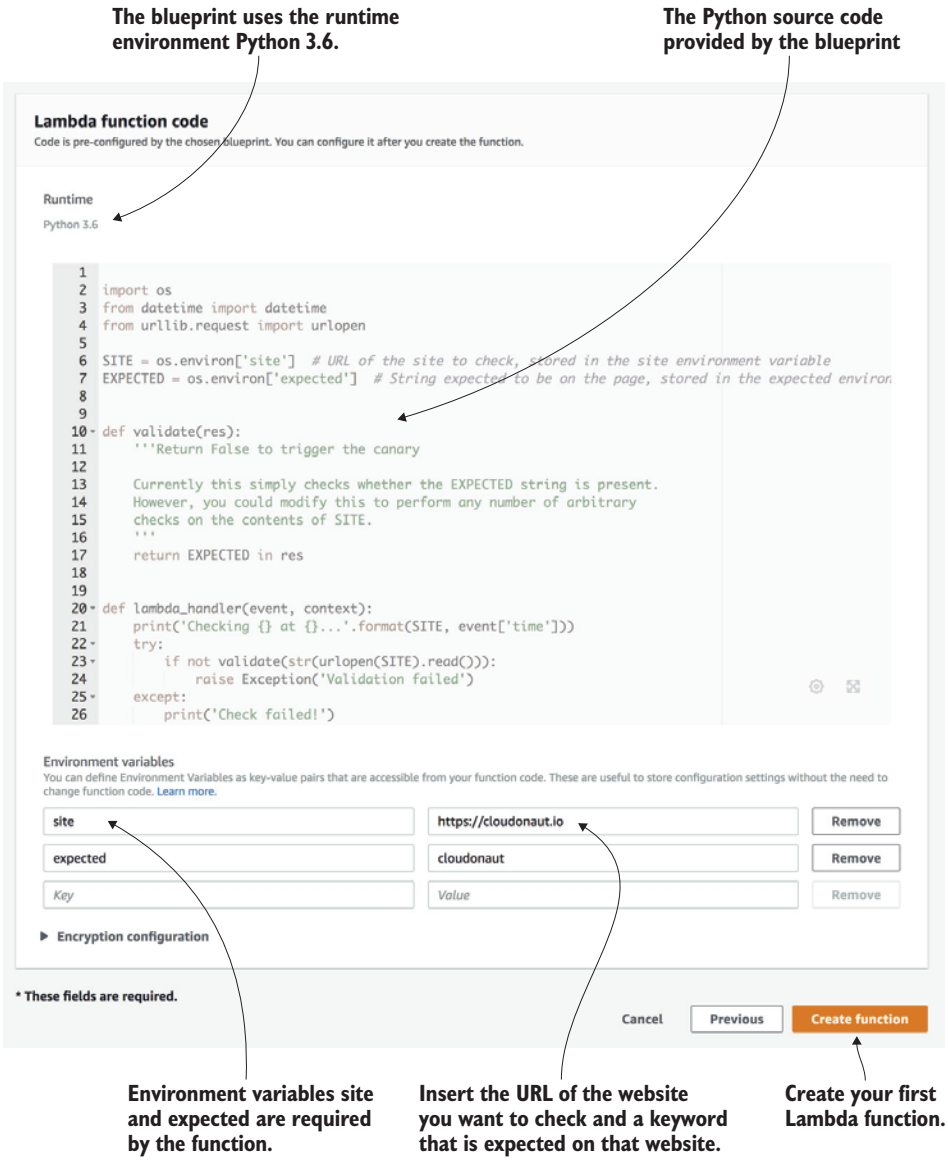


Figure 7.8 The predefined code implementing the website health check and environment variables to pass settings to the Lambda function

Congratulations, you have successfully created a Lambda function. Every 5 minutes, the function is invoked automatically and executes a health check for your website. Next, you will learn how to monitor your Lambda function and get notified via email whenever the health check fails.

7.2.2 Use CloudWatch to search through your Lambda function's logs

How do you know whether your website health check is working correctly? How do you even know if your Lambda function has been executed? It is time to look at how to monitor a Lambda function. You will learn how to access your Lambda function's log messages first. Afterward, you will create an alarm notifying you if your function fails.

Open the Monitoring tab in the details view of your Lambda function. You will find a chart illustrating the number of times your function has been invoked. Use the Reload button of the chart after a few minutes, if the chart isn't showing any invocations. To go to your Lambda function's logs, click View logs in CloudWatch as shown in figure 7.9.

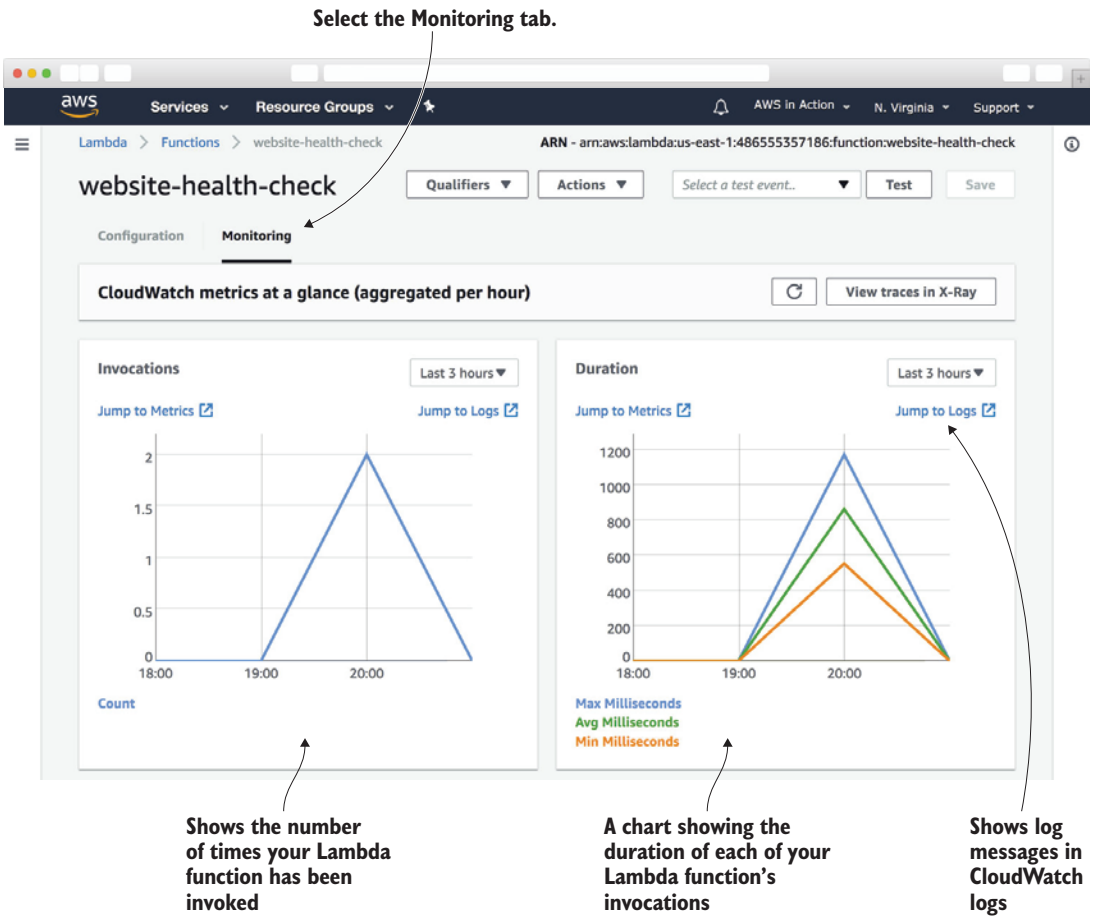


Figure 7.9 Monitoring overview: get insights into your Lambda function's invocations.

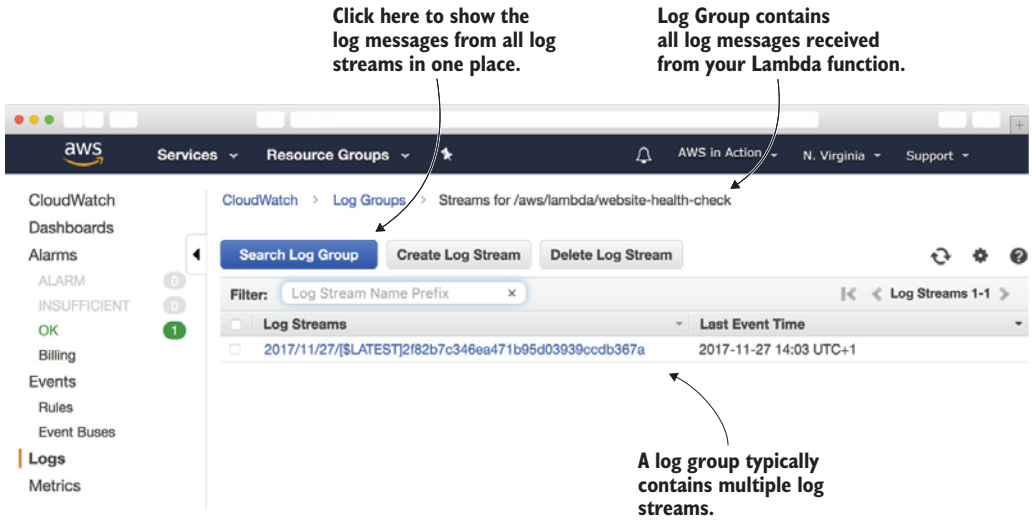


Figure 7.10 A log group collects log messages from a Lambda function stored in multiple log streams.

By default, your Lambda function sends log messages to CloudWatch. Figure 7.10 shows the log group named `/aws/lambda/website-health-check` that was created automatically and collects the logs from your function. Typically, a log group contains multiple log streams, allowing the log group to scale. Click Search Log Group to view the log messages from all streams in one view.

All log messages are presented in the overview of log streams, as shown in figure 7.11. You should be able to find a log message `Check passed!` indicating that the website health check was executed and passed successfully, for example.

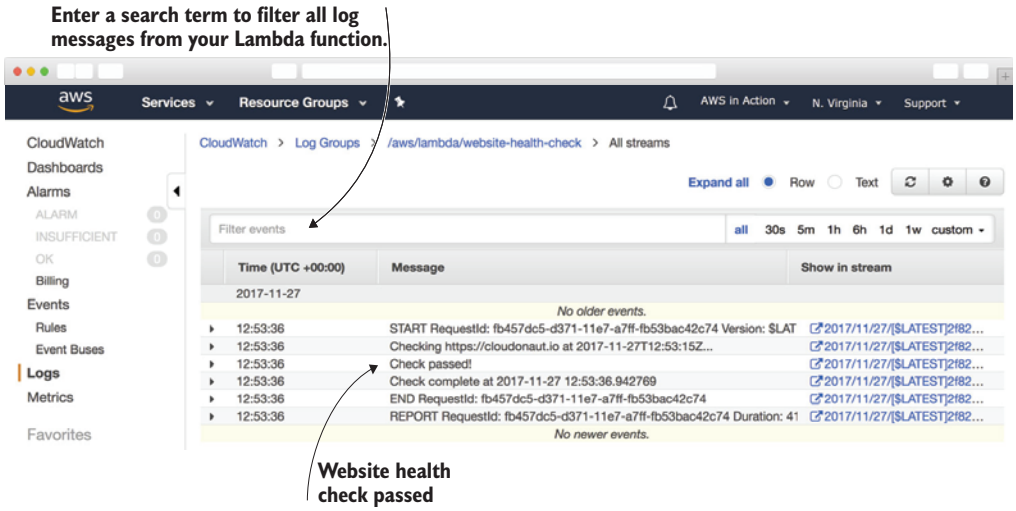


Figure 7.11 CloudWatch shows the log messages of your Lambda function.

The log messages show up after a delay of a few minutes. Reload the table if you are missing any log messages.

Being able to search through log messages in a centralized place is handy when debugging Lambda functions, especially if you are writing your own code. When using Python, you can use `print` statements or use the `logging` module to send log messages to CloudWatch out-of-the-box.

7.2.3 Monitoring a Lambda function with CloudWatch metrics and alarms

The Lambda function checks the health of your website every 5 minutes. A log message with the result of each health check is written to CloudWatch. But how do you get notified via email if the health check fails? Each Lambda function publishes the metrics listed in table 7.2 to CloudWatch by default:

Table 7.2 The CloudWatch metrics published by each Lambda function

Name	Description
Invocations	Counts the number of times a function is invoked. Includes successful and failed invocations.
Errors	Counts the number of times a function failed due to errors inside the function. For example, exceptions or timeouts.
Duration	Measures how long the code takes to run, from the time when the code starts executing to when it stops executing.
Throttles	As discussed at the beginning of the chapter, there is a limit for how many copies of your Lambda function are running at one time. This metric counts how many invocations have been throttled due to reaching this limit. Contact AWS support to increase the limit if needed.

Whenever the website health check fails, the Lambda function returns an error, increasing the count of the Errors metric. You will create an alarm notifying you via email whenever this metric counts more than 0 errors. In general, we recommend to create an alarm on the following metrics to monitor your Lambda functions: Errors and Throttles.

The following steps guide you through creating a CloudWatch alarm to monitor your website health checks. Your Management Console still shows the CloudWatch service. Select Alarms from the sub-navigation menu. Did you create a billing alarm in chapter 1? If so, the alarm will be listed here. Next, click Create Alarm as shown in figure 7.12.

First of all, you need to select the Errors metric for your Lambda function. Click By Function Name under Lambda. Figure 7.13 shows an overview.

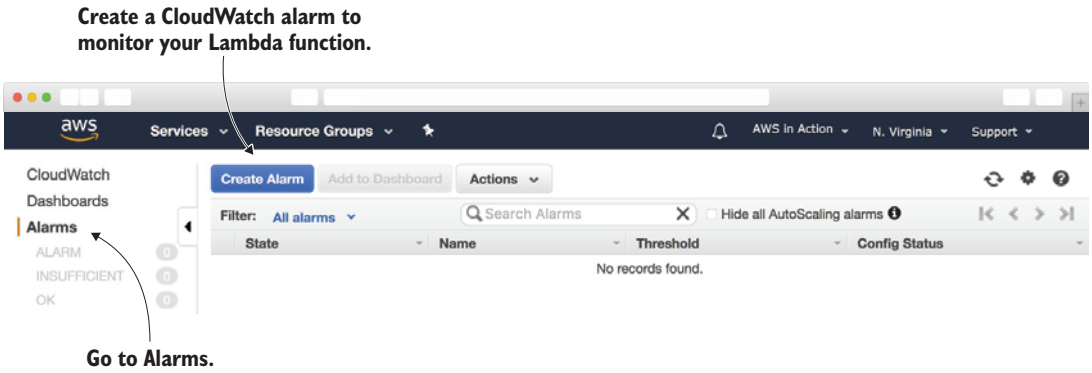


Figure 7.12 Starting the wizard to create a CloudWatch alarm to monitor a Lambda function

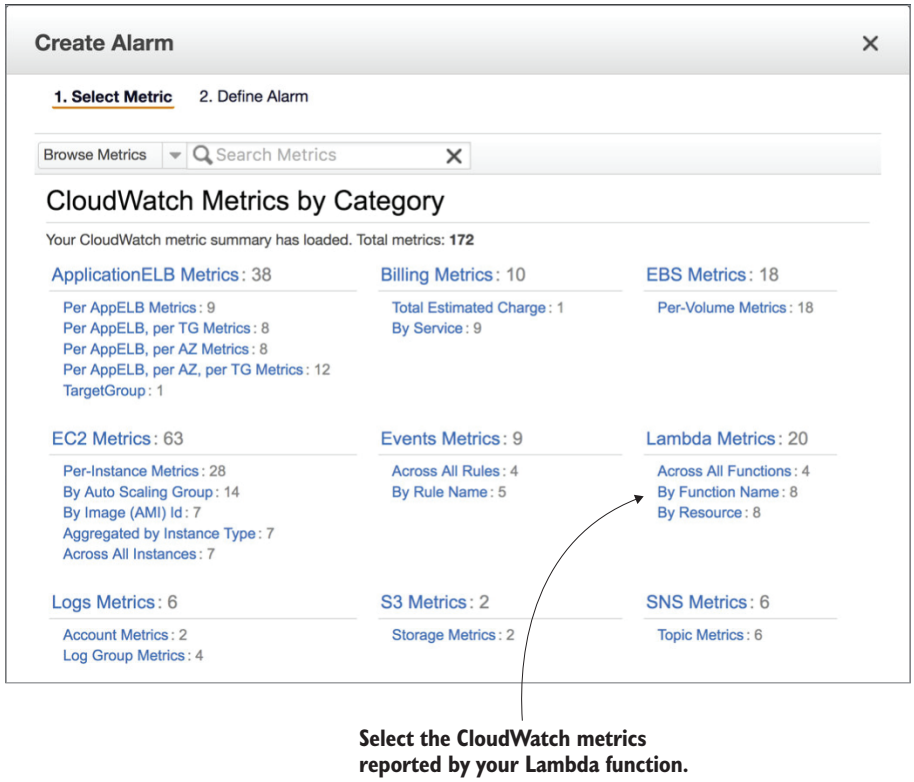


Figure 7.13 Selecting the Lambda metric to create an alarm

The next step is to select and prepare the metric for the alarm. Figure 7.14 shows the necessary steps:

1. Search for metrics belonging to your Lambda function `website-health-check`.
2. Select the Errors metric.
3. For the statistics, use Sum. This will set the alarm to trigger based on the total number of errors.
4. Choose 5 minutes for how often the alarm is updated (the time bucket).
5. Select a timespan from 5 minutes ago to now.
6. Click the Next button to proceed with the next step.

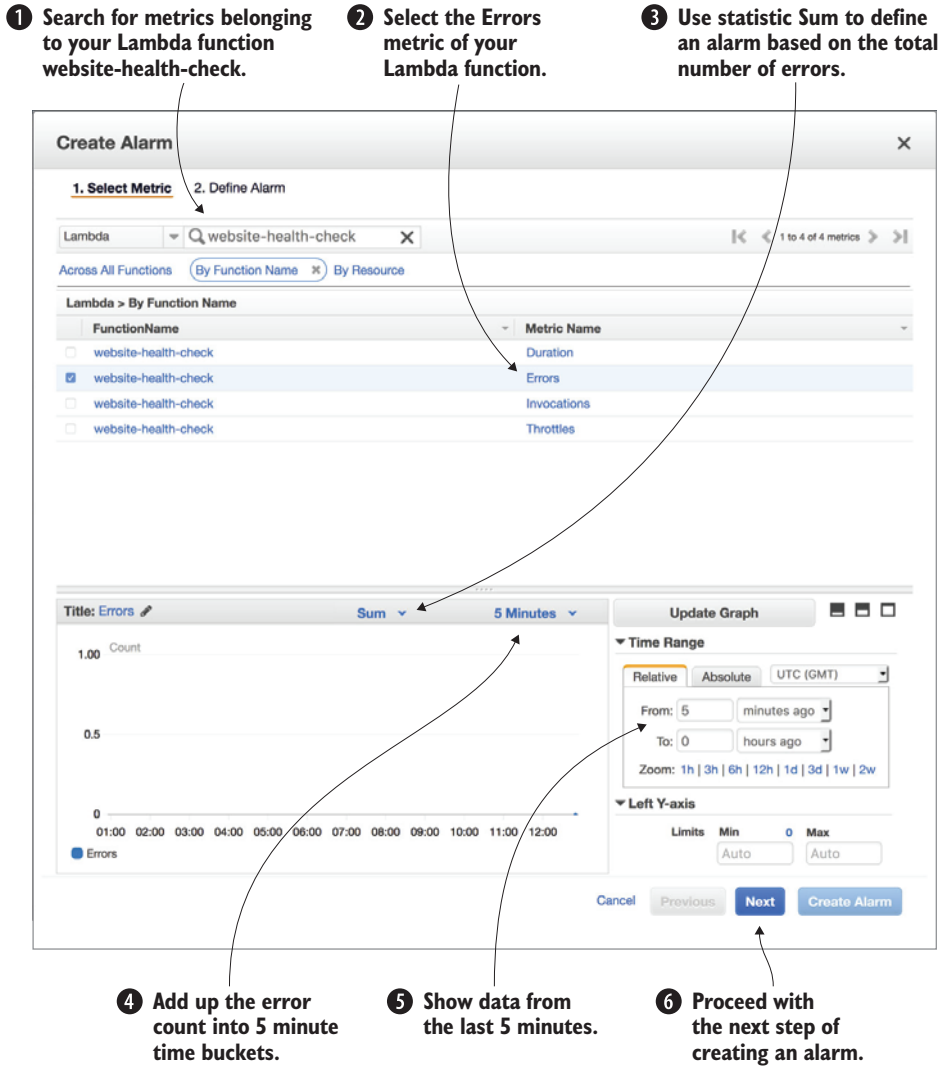


Figure 7.14 Selecting and preparing the metric view for the alarm

To create the alarm you need to define a name, a threshold, and the actions to be performed. Figure 7.15 shows the details.

1. Type in `website-health-check-error` as Name for the alarm.
2. Define a Description for the alarm.
3. Specify a threshold for the alarm. Use the drop-down boxes to set up the alarm like this: `Whenever Errors is > 0 for 1 out of 1 datapoints.`
4. Keep the default State is ALARM for the action.
5. Create a new notification list by clicking New List and typing in `website-health-check` as the name for the list.
6. Enter your email into the Email list.
7. Click the Create Alarm button.

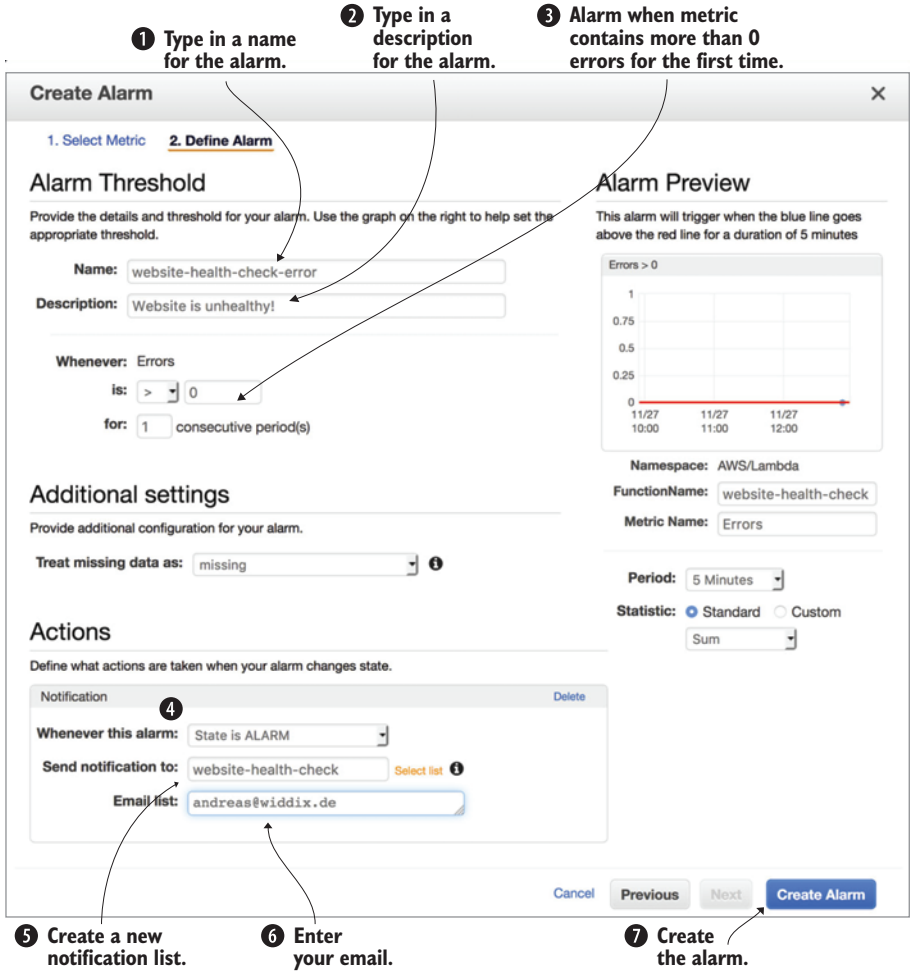


Figure 7.15 Creating an alarm by defining a threshold and defining an alarm action to send notifications via email

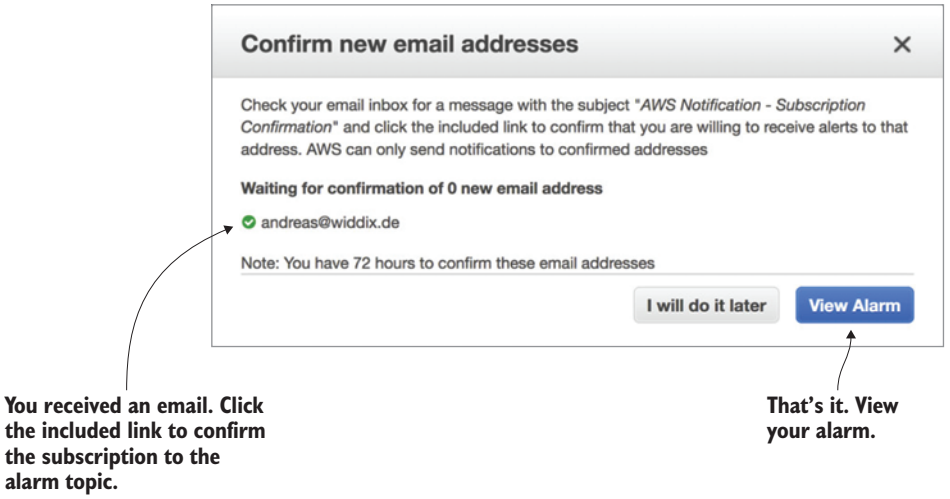


Figure 7.16 Check your inbox and confirm your subscription to the notification list.

As shown in figure 7.16, you will be sent an email including a confirmation link. Check your inbox and click the link to confirm your subscription to the notification list. Afterward, click the View Alarm button.

You will receive an alarm via email whenever your website health check fails. As you probably do not want to take down your website just so you can check that the alarm, you could change the function's environment variables. For example, you could change the `expected` value to a text snippet that your website does not include. A few minutes after changing the environment variable, you will receive an alarm via email.

Cleaning up

Open your Management Console and follow these steps to delete all the resources you have created during this section.

1. Go to the AWS Lambda service and delete the function named `website-health-check`.
2. Open the AWS CloudWatch service, select Logs from the sub-navigation menu, and delete the log group `/aws/lambda/website-health-check`.
3. Go to the AWS CloudWatch service, select Events from the sub-navigation menu, and delete the rule `website-health-check`.
4. Select Alarms from the sub navigation, and delete the alarm `website-health-check-error`.
5. Jump to the AWS IAM service, select Roles from the sub-navigation menu, and delete the role `lambda_basic_execution`.

7.2.4 Accessing endpoints within a VPC

As illustrated in figure 7.17, Lambda functions run outside your networks defined with VPC by default. However, Lambda functions are connected to the internet and therefore able to access other services. That's exactly what you have been doing when creating a website health check: the Lambda function was sending HTTP requests over the internet.

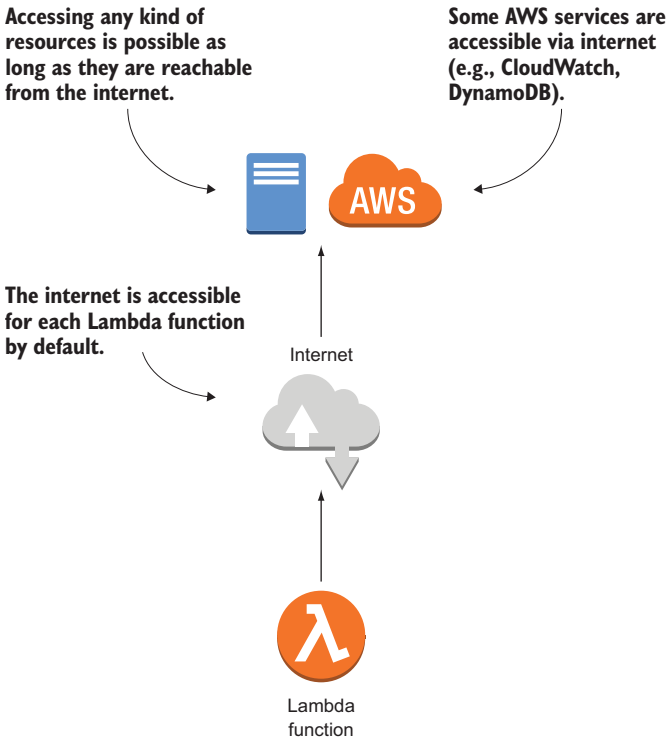


Figure 7.17 By default a Lambda function is connected to the internet and running outside your VPCs.

So what do you do when you have to reach a resource running in a private network within your VPC? For example, if you want to run a health check for an internal website? If you add network interfaces to your Lambda function, the function can access resources within your VPCs as shown in figure 7.18.

To do so you have to define the VPC, the subnets, as well as security groups for your Lambda function. See "Configuring a Lambda Function to Access Resources in an Amazon VPC" at http://docs.aws.amazon.com/lambda/latest/dg/vpc.html for more details. We have been using the ability to access resources within a VPC to access databases in various projects.

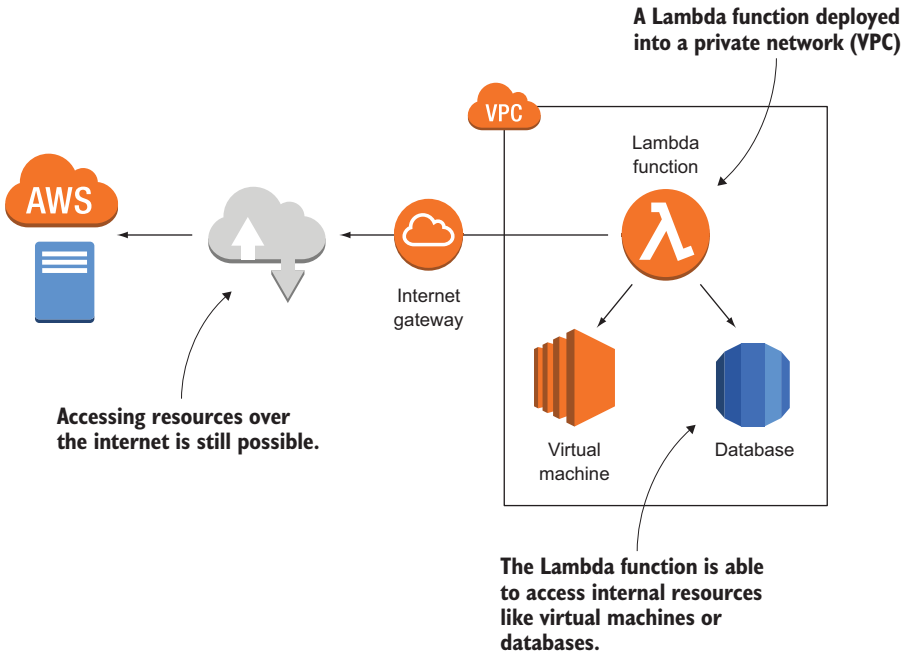


Figure 7.18 Deploying a Lambda function into your VPC allows you to access internal resources (such as database, virtual machines, and so on).

AWS recommends placing a Lambda function in a VPC only when absolutely necessary to reach a resource that is not accessible otherwise. Placing a Lambda function in a VPC increases complexity, especially when scaling to a large number of concurrent executions. For example, the number of available private IP addresses in a VPC is limited, but a Lambda function will need multiple private IP addresses to be able to scale the number of concurrent invocations.

7.3 Adding a tag containing the owner of an EC2 instance automatically

After using one of AWS's predefined blueprints to create a Lambda function, you will implement a Lambda function from scratch in this section. We are strongly focused on setting up your cloud infrastructure in an automated way. That's why you will learn how to deploy a Lambda function and all its dependencies without needing the Management Console.

Are you working in an AWS account together with your colleagues? Have you ever wondered who launched a certain EC2 instance? Sometimes you need to find out the owner of an EC2 instance for the following reasons:

- Double-checking if it is safe to terminate an unused instance without losing relevant data.
- Reviewing changes to an instance's configuration with its owner (such as making changes to the firewall configuration).
- Attributing costs to individuals, projects, or departments.
- Restricting access to an instance (for example, so only the owner is allowed to terminate an instance).

Adding a tag that states who owns an instance solves all these use cases. A tag can be added to an EC2 instance or almost any other AWS resource, and consists of a key and a value. Tags can be used to add information to a resource, filter resources, attribute costs to resources, as well as to restrict access. See "Tagging Your Amazon EC2 Resources" at http://mng.bz/pEJN for more details.

It is possible to add tags specifying the owner of an EC2 instance manually. But sooner or later, someone will forget to add the owner tag. There is a better solution for that! You will implement and deploy a Lambda function that adds a tag containing the name of the user who launched an EC2 instance automatically in the following section. But how do you execute a Lambda function every time an EC2 instance is launched, so that you can add the tag?

7.3.1 Event-driven: Subscribing to CloudWatch events

CloudWatch consists of multiple parts. You have already learned about metrics, alarms, and logs during this chapter. Another feature built into CloudWatch is called *events*. Whenever something changes in your infrastructure, an event is generated in near real-time:

- CloudTrail emits an event for every call to the AWS API.
- EC2 emits events whenever the state of an EC2 instances changes (such as when the state changes from pending to running).
- AWS emits an event to notify you of service degradations or downtimes.

Whenever you launch a new EC2 instance, you are sending a call to the AWS API. Subsequently, CloudTrail generates a CloudWatch event. Our goal is to add a tag to every new EC2 instance. Therefore, we are executing a function for every event that indicates the launch of a new EC2 instance. To trigger a Lambda function whenever such an event occurs, you need a rule. As illustrated in figure 7.19, the rule matches incoming events and routes them to a target, a Lambda function in our case.

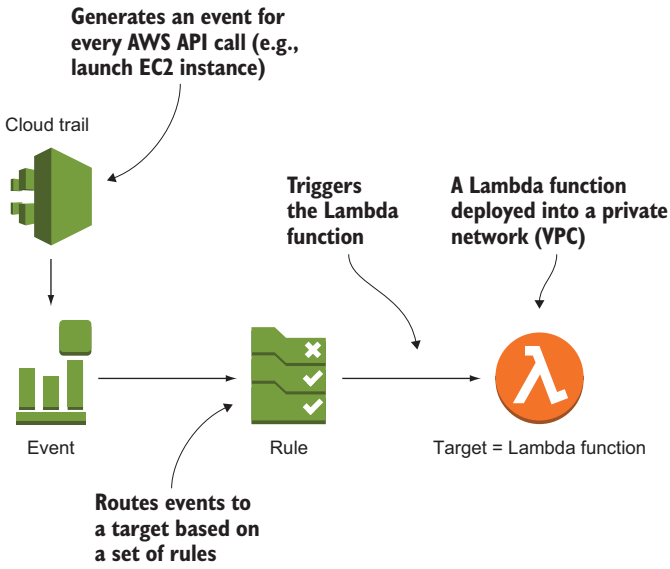


Figure 7.19 CloudTrail generates an event for every AWS API call, a rule routes the event to the Lambda function

Listing 7.1 shows some of the event details generated by CloudTrail whenever someone launches an EC2 instance. For our case, we're interested in the following information:

- `detail-type`—The event has been created by CloudTrail.
- `source`—The EC2 service is the source of the event.
- `eventName`—The event name `RunInstances` indicates that the event was generated because of an AWS API call launching an EC2 instance.
- `userIdentity`—Who called the AWS API to launch an instance?
- `responseElements`—The response from the AWS API when launching an instance. This includes the ID of the launched EC2 instance that we will need to add a tag to the instance later.

Listing 7.1 CloudWatch event generated by CloudTrail when launching an EC2 instance

```
{
  "version": "0",
  "id": "8a50bfef-33fd-2ea3-1056-02ad1eac7210",
  "detail-type": "AWS API Call via CloudTrail",
  "source": "aws.ec2",
  "account": "XXXXXXXXXXX",
  "time": "2017-11-30T09:51:25Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "eventVersion": "1.05",
    "userIdentity": {
```

CloudTrail generated the event

Someone sent a call to the AWS API affecting the EC2 service.

Information about the user who launched the EC2 instance

```
      "type": "IAMUser",
      "principalId": "...",
      "arn": "arn:aws:iam::XXXXXXXXXXXX:user/myuser",
      "accountId": "XXXXXXXXXXXX",
      "accessKeyId": "...",
      "userName": "myuser",
      "sessionContext": {
        "attributes": {
          "mfaAuthenticated": "false",
          "creationDate": "2017-11-30T09:51:05Z"
        }
      },
      "invokedBy": "signin.amazonaws.com"
    },
    "eventTime": "2017-11-30T09:51:25Z",
    "eventSource": "ec2.amazonaws.com",
    "eventName": "RunInstances",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "XXX.XXX.XXX.XXX",
    "userAgent": "signin.amazonaws.com",
    "requestParameters": {
      [...]
    },
    "responseElements": {
      "requestId": "327f5231-c65a-468c-83a8-b00b7c949f78",
      "reservationId": "r-0234df5d03e3ad6a5",
      "ownerId": "XXXXXXXXXXXX",
      "groupSet": {},
      "instancesSet": {
        "items": [
          {
            "instanceId": "i-06133867ab0f704e7",
            "imageId": "ami-55ef662f",
            [...]
          }
        ]
      }
    },
    "requestID": "327f5231-c65a-468c-83a8-b00b7c949f78",
    "eventID": "134d35c6-6a76-49b9-9ff8-5a4130474b6f",
    "eventType": "AwsApiCall"
  }
}
```

ID of the user who launched the EC2 instance

Event was generated because a RunInstances call (used to launch an EC2 instance) was processed by the AWS API.

Response of the AWS API when launching the instance

ID of the launched EC2 instance

A rule consists of an event pattern for selecting events, along with a definition of one or multiple targets. The following pattern selects all events from CloudTrail generated by an AWS API call affecting the EC2 service. The pattern matches three attributes from the event described in listing 7.1: `detail-type`, `source`, and `eventName`.

Listing 7.2 Rule to filter events from CloudTrail

```
{
  "detail-type": [
    "AWS API Call via CloudTrail"
```

Filter events from CloudTrail caused by AWS API calls.

```
  ],
  "source": [
    "aws.ec2"
  ],
  "detail": {
    "eventName": [
      "RunInstances"
    ]
  }
}
```

Filter events from the EC2 service.

Filter events with event name RunInstances, which is the AWS API call to launch an EC2 instance.

Defining filters on other event attributes is possible as well, in case you are planning to write another rule in the future. The rule format stays the same.

When specifying an event pattern, we typically use the following fields, which are included in every event:

- `source`—The namespace of the service which generated the event. See "Amazon Resource Names (ARNs) and AWS Service Namespaces" at http://mng.bz/GFGm for details.
- `detail-type`—Categorizes the event in more detail.

See "Event Patterns in CloudWatch Events" at http://mng.bz/37Ux for more detailed information.

You have now defined the events that will trigger your Lambda function. Next, you will implement the Lambda function.

7.3.2 Implementing the Lambda function in Python

Implementing the Lambda function to tag an EC2 instance with the owner's user name is simple. You will need to write no more than 10 lines of Python code. The programming model for a Lambda function depends on the programming language you choose. Although we are using Python in our example, you will be able to apply what you've learned when implementing a Lambda function in Java, Node.js, C#, or Go. As shown in the next listing, your function written in Python needs to implement a well-defined structure.

Listing 7.3 Lambda function written in Python

The name of the Python function, which is referenced by the AWS Lambda as the function handler. The event parameter is used to pass the CloudWatch event and the context parameter includes runtime information.

It is your job to implement the function.

```
def lambda_handler(event, context):
  # Insert your code
  return
```

Use return to end the function execution. It is not useful to hand over a value in this scenario, as the Lambda function is invoked asynchronously by a CloudWatch event.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. Switch to the `chapter07` directory, which includes all files needed for this example.

Time to write some Python code! The following listing for `lambda_function.py` shows the function, which receives an event from CloudTrail indicating that an EC2 has been launched recently, and adds a tag including the name of the instance's owner. The AWS SDK for Python 3.6, named `boto3`, is provided out-of-the-box in the Lambda runtime environment for Python 3.6. In this example you are using the AWS SDK to create a tag for the EC2 instance `ec2.create_tags(...)`. See the Boto 3 Documentation at https://boto3.readthedocs.io/en/latest/index.html if you are interested in the details of `boto3`.

Listing 7.4 Lambda function adding a tag to EC2 instance

Creates an AWS SDK client to manage the EC2 service

The name of the function used as entry point for the Lambda function

Extracts the user's name from the CloudTrail event

Extracts the instance's ID from the CloudTrail event

Adds a tag to the EC2 instance using the key owner and the user's name as value

```
import boto3
ec2 = boto3.client('ec2')

def lambda_handler(event, context):

  userName = event['detail']['userIdentity']['arn'].split('/')[1]
  instanceId = event['detail']['responseElements']
➥ ['instancesSet']['items'][0]['instanceId']
  print("Adding owner tag " + userName + " to instance " + instanceId + ".")
  ec2.create_tags(Resources=[instanceId,],
➥ Tags=[{'Key': 'Owner', 'Value': userName},])
  return
```

After implementing your function in Python, the next step is to deploy the Lambda function with all its dependencies.

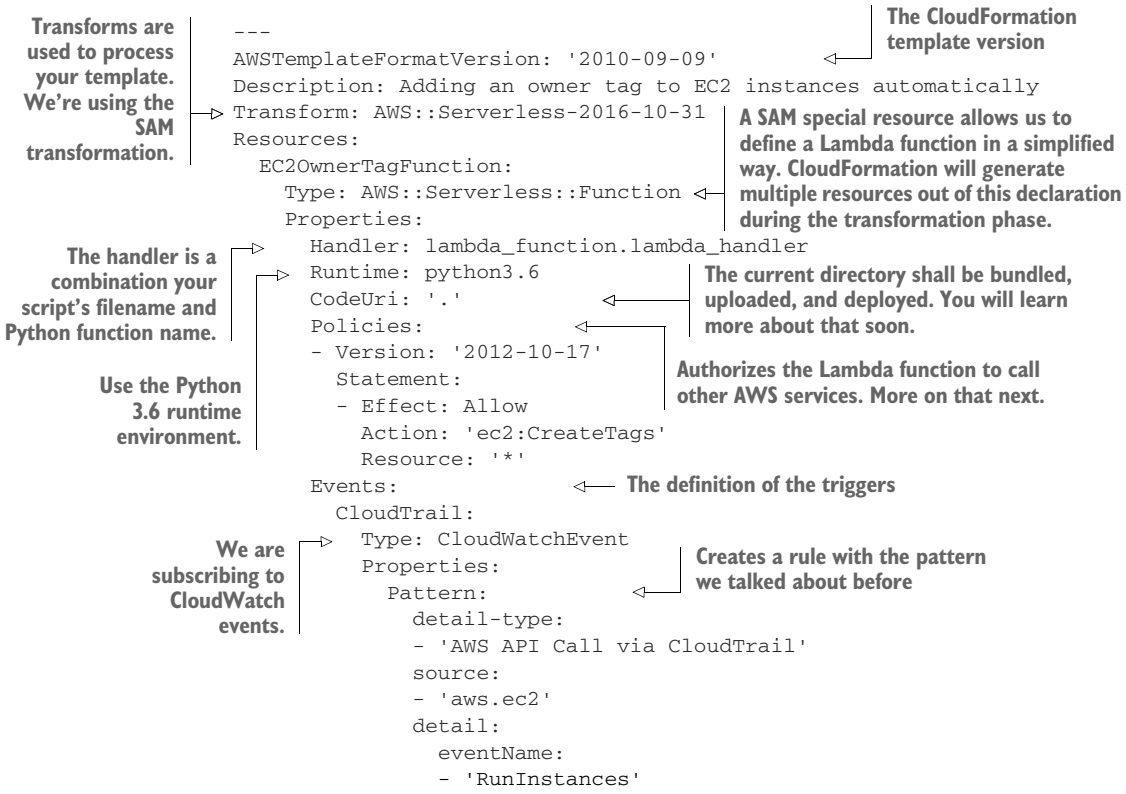
7.3.3 Setting up a Lambda function with the Serverless Application Model (SAM)

You have probably noticed that we are huge fans of automating infrastructures with CloudFormation. Using the Management Console is a perfect way to take the first step when learning about a new service on AWS. But leveling up from manually clicking through a web interface to fully automating the deployment of your infrastructure should be your second step.

AWS released the *Serverless Application Model* (SAM) in 2016. SAM provides a framework for serverless applications, extending plain CloudFormation templates to make it easier to deploy Lambda functions.

This listing shows how to define a Lambda function using SAM and a CloudFormation template.

Listing 7.5 Defining a Lambda function with SAM within a CloudFormation template



7.3.4 Authorizing a Lambda function to use other AWS services with an IAM role

Lambda functions typically interact with other AWS services. For instance, they might write log messages to CloudWatch allowing you to monitor and debug your Lambda function. Or they might create a tag for an EC2 instance, as in the current example. Therefore, calls to the AWS APIs need to be authenticated and authorized. Figure 7.20 shows a Lambda function assuming an IAM role to be able to send authenticated and authorized requests to other AWS services.

Temporary credentials are generated based on the IAM role and injected into each invocation via environment variables (such as `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`). Those environment variables are used by the AWS SDK to sign requests automatically.

You should follow the least-privilege principle: your function should only be allowed to access services and actions that are needed to perform the function's task.

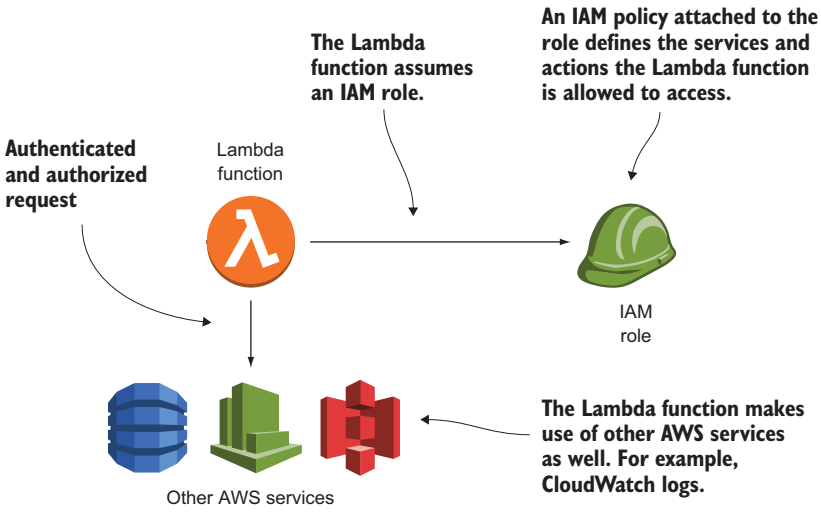


Figure 7.20 A Lambda function assumes an IAM role to authenticate and authorize requests to other AWS services.

You should specify a detailed IAM policy granting access to specific actions and resources.

Listing 7.6 shows an excerpt from the Lambda function's CloudFormation template based on SAM. When using SAM, an IAM role is created for each Lambda function by default. A managed policy that grants write access to CloudWatch logs is attached to the IAM role by default as well. Doing so allows the Lambda function to write to CloudWatch logs.

So far the Lambda function is not allowed to create a tag for the EC2 instance. You need a custom policy granting access to the `ec2:CreateTags`.

Listing 7.6 A custom policy for adding tags to EC2 instances

```
# [...]
EC2OwnerTagFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: lambda_function.lambda_handler
    Runtime: python3.6
    CodeUri: '.'
    Policies:
    - Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Action: 'ec2:CreateTags'
        Resource: '*'
    # [...]
```

Lets you define your own custom IAM policy that will be attached to the Lambda function's IAM role

The statement allows ...

...creating tags

...for all resources.

In case you implement another Lambda function in the future, make sure you create an IAM role granting access to all the services your function needs to access (such as reading objects from S3, writing data to a DynamoDB database, and so on). Revisit section 6.3 if you want to recap the details of IAM.

7.3.5 Deploying a Lambda function with SAM

To deploy a Lambda function, you need to upload the deployment package to S3. The deployment package is a zip file including your code as well as additional modules. Afterward, you need to create and configure the Lambda function as well as all the dependencies (the IAM role, event rule, and so on). Using SAM in combination with the AWS CLI allows you to accomplish both tasks.

First of all you, need to create an S3 bucket to store your deployment packages. Use the following command, replacing *$yourname* with your name to avoid name conflicts with other readers.

```
$ aws s3 mb s3://ec2-owner-tag-$yourname
```

The next step is to create the deployment package and upload the package to S3. Execute the following command in your terminal to do so. A copy of your template is stored as `output.yaml`, with a reference to the deployment package uploaded to S3.

```
$ aws cloudformation package --template-file template.yaml \
➥ --s3-bucket ec2-owner-tag-$yourname --output-template-file output.yaml
```

By typing in the following command in your terminal, you are deploying the Lambda function. This results in a CloudFormation stack named `ec2-owner-tag`. Make sure your working directory is the code directory `chapter07` containing the `template.yaml` and `lambda_function.py` files.

```
$ aws cloudformation deploy --stack-name ec2-owner-tag \
➥ --template-file output.yaml --capabilities CAPABILITY_IAM
```

You are a genius! Your Lambda function is up and running. Launch an EC2 instance and you will find a tag with your user name `myuser` attached after a few minutes.

Cleaning up

If you have launched an EC2 instance to test your Lambda function, don't forget to terminate the instance afterward.

Otherwise it is quite simple to delete the Lambda function and all its dependencies. Just execute the following command in your terminal. Replace *$yourname* with your name.

```
$ aws cloudformation delete-stack --stack-name ec2-owner-tag
$ aws s3 rb s3://ec2-owner-tag-$yourname --force
```

7.4 What else can you do with AWS Lambda?

In the last part of the chapter, we would like to share what else is possible with AWS Lambda, starting with Lambda's limitations and insights into the serverless pricing model. We will end with three use cases for serverless applications we have built for our consulting clients.

7.4.1 What are the limitations of AWS Lambda?

Each invocation of your Lambda function needs to complete within a maximum of 300 seconds. This means the problem you are solving with your function needs to be small enough to fit into the 300-second limit. It is probably not possible to download 10 GB of data from S3, process the data, and insert parts of the data into a database within a single invocation of a Lambda function. But even if your use case fits into the 300-second constraint, make sure that it does under all circumstances. Here's a short anecdote from one of our first serverless projects: We built a serverless application that pre-processed analytics data from news sites. The Lambda functions typically processed the data within less than 180 seconds. But when the 2017 U.S. elections came, the volume of the analytics data exploded in a way no one expected. Our Lambda functions were no longer able to complete within 300 seconds. A show stopper for our serverless approach.

AWS Lambda provisions and manages the resources needed to run your function. A new execution context is created in the background every time you deploy a new version of your code, go a long time without any invocations, or when the number of concurrent invocations increases. Starting a new execution context requires AWS Lambda to download your code, initialize a runtime environment, and load your code. This process is called a *cold-start*. Depending on the size of your deployment package, the runtime environment, and your configuration, a cold-start could take from a few milliseconds to a few seconds. Therefore, applications with very strict requirements concerning response times are not good candidates for AWS Lambda. Conversely, there are a lot of use cases where the additional latency caused by a cold-start is acceptable. For example, the two examples in this chapter are not affected by a cold-start at all. To minimize cold-start times, you should keep the size of your deployment package as small as possible, provision additional memory, and use a runtime environment like Python, Node.js, or Go instead of C# or Java.

Another limitation is the maximum amount of memory you can provision for a Lambda function: 3008 MB. If your Lambda function uses more memory, its execution will be terminated.

It is also important to know that CPU and networking capacity are allocated to a Lambda function based on the provisioned memory as well. So if you are running computing or network-intensive work within a Lambda function, increasing the provisioned memory will probably improve performance.

At the same time, the default limit for the maximum size of the compressed deployment package (zip file) is 50 MB. When executing your Lambda function, you can use up to 500 MB non-persistent disk space mounted to `/tmp`.

Look at "AWS Lambda Limits" at http://docs.aws.amazon.com/lambda/latest/dg/limits.html if you want to learn more about Lambda's limitations.

7.4.2 Impacts of the serverless pricing model

When launching a virtual machine, you have to pay AWS for every operating hour, billed in second-intervals. You are paying for the machines no matter if you are using the resource they provide. Even when nobody is accessing your website or using your application, you are paying for the virtual machine.

That's totally different with AWS Lambda. Lambda is billed per request. Costs occur only when someone accesses your website or uses your application. That's a game changer, especially for applications with uneven access patterns, or for applications that are used rarely. Table 7.3 explains the Lambda pricing model in detail.

Table 7.3 AWS Lambda pricing model

	Free Tier	Occurring costs
Number of Lambda function invocations	First 1 million requests every month	$0.0000002 USD per request
Duration billed in 100 ms increments based on the amount of memory you provisioned for your Lambda function	Using the equivalent of 400,000 seconds of a Lambda function with 1 GB provisioned memory every month	$0.00001667 USD for using 1 GB for one second

Free Tier for AWS Lambda

The Free Tier for AWS Lambda does not expire after 12 months. That's a huge difference compared to the Free Tier of other AWS services (such as EC2) where you are only eligible for the Free Tier within the first 12 month after creating an AWS account.

Sounds complicated? Figure 7.21 shows an excerpt of an AWS bill. The bill is from November 2017, and belongs to an AWS account we are using to run a chatbot (see marbot.io). Our chatbot implementation is 100% serverless. The Lambda functions were executed 1.2 million times in November 2017, which results in a charge of $0.04 USD. All our Lambda functions are configured to provision 1536 MB memory. In total, all our Lambda functions have been running for 216,000 seconds, roundabout 60 hours in November 2017. That's still within the Free Tier of 400,000 seconds with 1 GB provisioned memory every month. So in total we had to pay $0.04 for using AWS Lambda in November 2017, which allowed us to serve around 400 customers with our chatbot.

▾ Lambda		$0.04
▾ US East (Northern Virginia) Region		$0.04
AWS Lambda Lambda-GB-Second		$0.00
AWS Lambda - Compute Free Tier - 400,000 GB-Seconds - US East (Northern Virginia)	331,906.500 seconds	$0.00
AWS Lambda Request		$0.04
0.0000000367 USD per AWS Lambda - Requests Free Tier - 1,000,000 Requests - US East (Northern Virginia) (blended price)*	1,209,096 Requests	$0.04

Figure 7.21 Excerpt from our AWS bill from November 2017 showing costs for AWS Lambda

This is only a small piece of our AWS bill, as other services used together with AWS Lambda, for example to store data, add more significant costs to our bill.

Don't forget to compare costs between AWS Lambda and EC2. Especially in a high-load scenario with more than 10 million requests per day, using AWS Lambda will probably result in higher costs compared to using EC2. But comparing infrastructure costs is only one part of what you should be looking at. Consider the total cost of ownership (TOC), including costs for managing your virtual machines, performing load and resilience tests, and automating deployments as well.

Our experience has shown that the total cost of ownership is typically lower when running an application on AWS Lambda compared to Amazon EC2.

The last part of the chapter focuses on additional use cases for AWS Lambda besides automating operational tasks, as you have done thus far.

7.4.3 Use case: Web application

A common use case for AWS Lambda is building a back end for a web or mobile application. As illustrated in figure 7.22, an architecture for a serverless web application typically consists of the following building blocks:

- *Amazon API Gateway*—Offers a scalable and secure REST API that accepts HTTPS requests from your web application's front end or mobile application.
- *AWS Lambda*—Lambda functions are triggered by the API gateway. Your Lambda function receives data from the request and returns the data for the response.
- *Object store and NoSQL database*—For storing and querying data, your Lambda functions typically use additional services offering object storage or NoSQL databases, for example.

Do you want to get started building web applications based on AWS Lambda? We recommend *AWS Lambda in Action* from Danilo Poccia, (Manning, 2016).

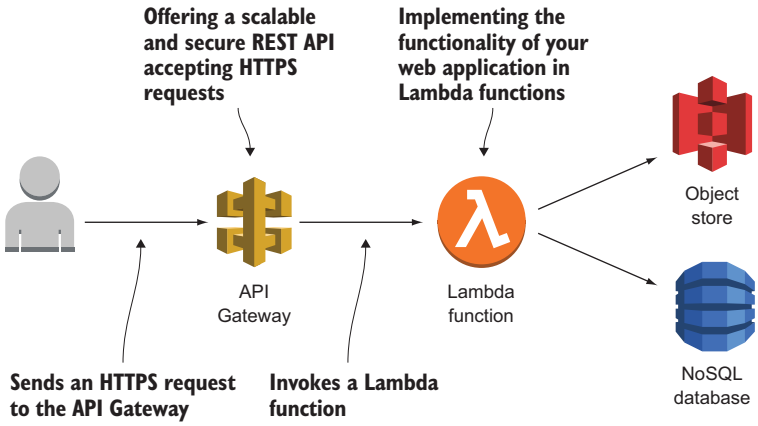


Figure 7.22 A web application build with API Gateway and Lambda

7.4.4 Use case: Data processing

Another popular use case for AWS Lambda is event-driven data processing. Whenever new data is available, an event is generated. The event triggers the data processing needed to extract or transform the data. Figure 7.23 shows an example.

1. The load balancer collects access logs and uploads them to an object store periodically.
2. Whenever an object is created or modified, the object store triggers a Lambda function automatically.
3. The Lambda function downloads the file including the access logs from the object store, and sends the data to an Elasticsearch database to be available for analytics.

We have successfully implemented this scenario in various projects. Keep in mind the maximum execution limit of 300 seconds when implementing data processing jobs with AWS Lambda.

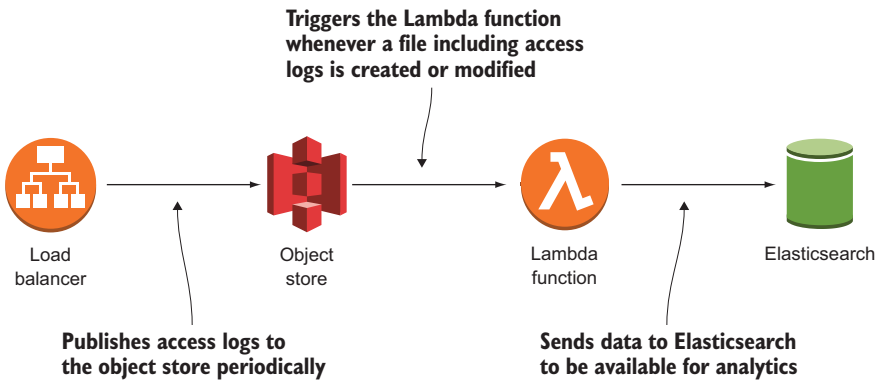


Figure 7.23 Processing access logs from a load balancer with AWS Lambda

7.4.5 Use case: IoT back end

The AWS IoT service provides building blocks needed to communicate with various devices (things) and build event-driven applications. Figure 7.24 shows an example. Each thing publishes sensor data to a message broker. A rule filters the relevant messages and triggers a Lambda function. The Lambda function processes the event and decides what steps are needed based on business logic you provide.

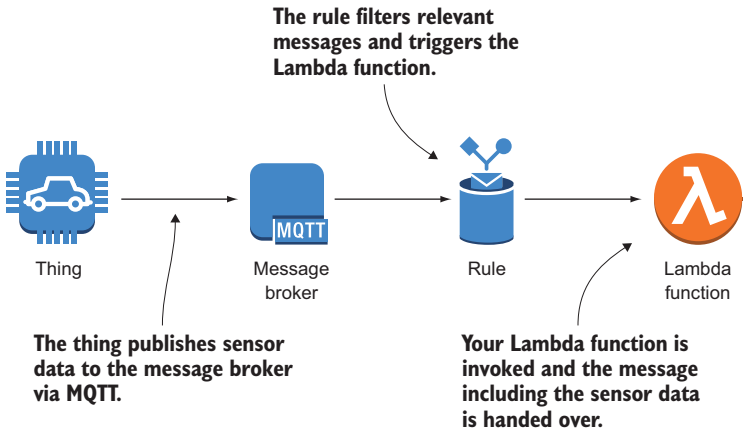


Figure 7.24 Processing access logs from a load balancer with AWS Lambda

We built a proof-of-concept for collecting sensor data and publishing metrics to a dashboard with AWS IoT and AWS Lambda, for example.

We have gone through three possible use cases for AWS Lambda, but we haven't covered all of them. AWS Lambda is integrated with many other services as well. If you want to learn much more about AWS Lambda, we recommend the following books:

- *AWS Lambda in Action* by Danilo Poccia (Manning, 2016) is an example-driven tutorial that teaches how to build applications using an event-based approach on the back end.
- *Serverless Architectures on AWS* by Peter Sbarski (Manning, 2017) teaches how to build, secure, and manage serverless architectures that can power the most demanding web and mobile apps.

Summary

- AWS Lambda allows you to run your Java, Node.js, C#, Python, and Go code within a fully managed, highly available, and scalable environment.
- The Management Console and blueprints offered by AWS help you to get started quickly.
- By using a schedule expression, you can trigger a Lambda function periodically. This is comparable to triggering a script with the help of a cron job.

- The Serverless Application Model (SAM) enables you to deploy a Lambda function in an automated way with AWS CloudFormation.
- There are many event sources for using Lambda functions in an event-driven way. For example, you can subscribe to events triggered by CloudTrail for every request you send to the AWS API.
- The most important limitation of a Lambda function is the maximum duration of 300 seconds per invocation.

Part 3

Storing data in the cloud

There is one guy called Singleton in your office who knows all about your file server. If Singleton is out of office, no one else can maintain the file server. As you can imagine, while Singleton is on holiday, the file server crashes. No one else knows where the backup is located, but the boss needs the document now or the company will lose a lot of money. If Singleton had stored his knowledge in a database, coworkers could look up the information. But because the knowledge and Singleton are tidily coupled, the information is unavailable.

Imagine a virtual machine where important files are located on hard disk. As long as the virtual machine is up and running, everything is fine. But everything fails all the time, including virtual machines. If a user uploads a document on your website, where is it stored? Chances are high that the document is persisted to hard disk on the virtual machine. Let's imagine that the document was uploaded to your website but persisted as an object in an independent object store. If the virtual machine fails, the document will still be available. If you need two virtual machines to handle the load on your website, they both have access to that document because it is not tightly coupled with a single virtual machine. If you separate your state from your virtual machine, you will be able to become fault-tolerant and elastic. Let highly specialized solutions like object stores and databases persist your state.

AWS offers many ways to store your data. The following table helps to decide which service to use for your data on a high level. The comparison is only a rough overview. We recommend that you choose 2–3 services that best fit your use case and then jump into the details by reading the chapters to make your decision.

Overview of data storage services

Service	Access	Maximum storage volume	Latency	Storage cost
S3	AWS API (SDKs, CLI), third party tools	unlimited	high	very low
Glacier	S3, AWS API (SDKs, CLI), third party tools	unlimited	extreme high	extreme low
EBS (SSD)	Attached to an EC2 instance via network	16 TB	low	low
EC2 Instance Store (SSD)	Attached to an EC2 instance directly	15 TB	very low	very low
EFS	NFSv4.1, for example from EC2 instance or on-premises	unlimited	medium	medium
RDS (MySQL, SSD)	SQL	6 TB	medium	low
ElastiCache	Redis / memcached protocol	6.5 TB	low	high
DynamoDB	AWS API (SDKs, CLI)	unlimited	medium	medium

Chapter 8 will introduce S3, a service offering object storage. You will learn how to integrate the object storage into your applications to implement a stateless server.

Chapter 9 is about block-level storage for virtual machines offered by AWS. You will learn how to operate legacy software on block-level storage.

Chapter 10 covers highly available block-level storage that can be shared across multiple virtual machines offered by AWS.

Chapter 11 introduces RDS, a service that offers managed relational database systems like PostgreSQL, MySQL, Oracle, or Microsoft SQL server. If your applications use such a relational database system, this is an easy way to implement a stateless server architecture.

Chapter 12 introduces ElastiCache, a service that offers managed in-memory database systems like Redis or Memcached. If your applications need to cache data, you can use an in-memory database to externalize ephemeral state.

Chapter 13 will introduce DynamoDB, a service that offers a NoSQL database. You can integrate this NoSQL database into your applications to implement a stateless server.

8 Storing your objects: S3 and Glacier

This chapter covers

- Transferring files to S3 using the terminal
- Integrating S3 into your applications with SDKs
- Hosting a static website with S3
- Diving into the internals of the S3 object store

Storing data comes with two challenges: ever-increasing volumes of data and ensuring durability. Solving the challenges is hard or even impossible if using disks connected to a single machine. For this reason, this chapter covers a revolutionary approach: a distributed data store consisting of a large number of machines connected over a network. This way, you can store near-unlimited amounts of data by adding additional machines to the distributed data store. And since your data is always stored on more than one machine, you reduce the risk of losing that data dramatically.

You will learn about how to store images, videos, documents, executables, or any other kind of data on Amazon S3 in this chapter. Amazon S3 is a simple-to-use, fully managed distributed data store provided by AWS. Data is managed as objects, so

the storage system is called an *object store*. We will show you how to use S3 to back up your data, how to integrate S3 into your own application for storing user-generated content, as well as how to host static websites on S3.

On top of that, we will introduce Amazon Glacier, a backup and archiving store. On one hand, storing data on Amazon Glacier costs less than storing data on Amazon S3. Conversely, retrieving data from Amazon Glacier takes up to 5 hours, compared to immediate access from Amazon S3.

Examples are 100% covered by Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

8.1 What is an object store?

Back in the old days, data was managed in a hierarchy consisting of folders and files. The file was the representation of the data. In an *object store*, data is stored as objects. Each object consists of a globally unique identifier, some metadata, and the data itself, as figure 8.1 illustrates. An object's *globally unique identifier* (GUID) is also known as its *key*; you can address the object from different devices and machines in a distributed system using the GUID.

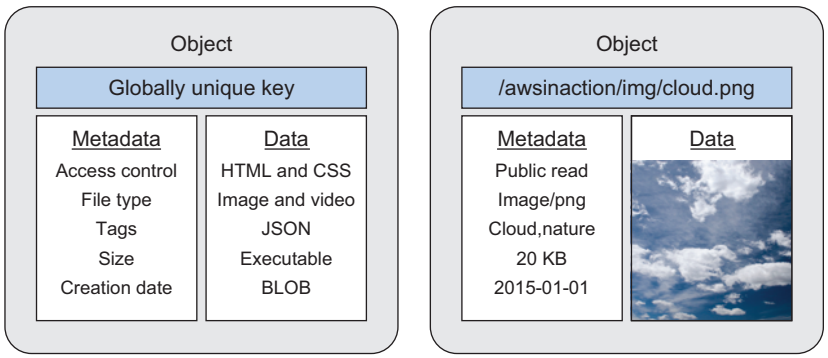


Figure 8.1 Objects stored in an object store have three parts: a unique ID, metadata describing the content, and the content itself (such as an image).

You can use metadata to enrich an object with additional information. Typical examples for object metadata are:

- Date of last modification
- Object size
- Object's owner
- Object's content type

It is possible to request only an object's metadata without requesting the data itself. This is useful if you want to list objects and their metadata before accessing a specific object's data.

8.2 Amazon S3

Amazon S3 is a distributed data store, and one of the oldest services provided by AWS. *Amazon S3* is an acronym for *Amazon Simple Storage Service*. It's a typical web service that lets you store and retrieve data organized as objects via an API reachable over HTTPS.

Here are some typical use cases:

- Storing and delivering static website content. For example, our blog cloudonaut.io is hosted on S3.
- Backing up data. For example, Andreas backs up his photo library from his computer to S3 using the AWS CLI.
- Storing structured data for analytics, also called a *data lake*. For example, I use S3 to store JSON files containing the results of performance benchmarks.
- Storing and delivering user-generated content. For example, I built a web application—with the help of the AWS SDK—that stores user uploads on S3.

Amazon S3 offers unlimited storage space, and stores your data in a highly available and durable way. You can store any kind of data, such as images, documents, and binaries, as long as the size of a single object doesn't exceed 5 TB. You have to pay for every GB you store in S3, and you also incur costs for every request and for all transferred data. As figure 8.2 shows, you can access S3 via the internet using HTTPS to upload and download objects. To access S3 you can use the Management Console, the CLI, SDKs, or third-party tools.

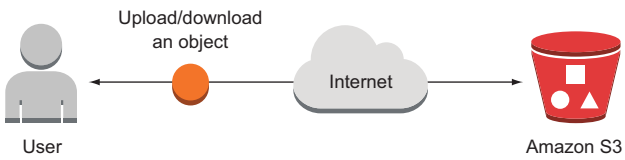


Figure 8.2 Uploading and downloading an object to S3 via HTTPS

S3 uses *buckets* to group objects. A bucket is a container for objects. You can create multiple buckets, each of which has a globally unique name, to separate data for different scenarios. By *unique* we really mean unique—you have to choose a bucket name that isn't used by any other AWS customer in any other region. Figure 8.3 shows the concept.

You will learn how to upload and download data to S3 using the AWS CLI next.

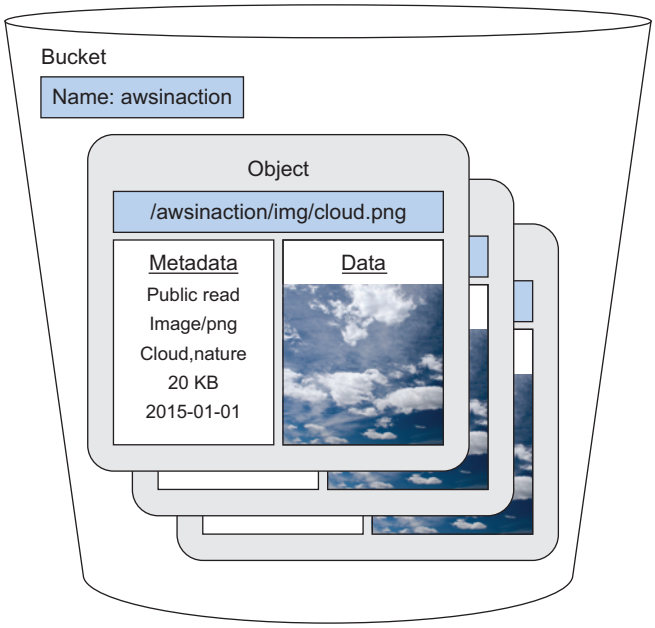


Figure 8.3 S3 uses buckets with a globally unique name to group objects.

8.3 Backing up your data on S3 with AWS CLI

Critical data needs to be backed up to avoid loss. Backing up data at an off-site location decreases the risk of losing data even during extreme conditions like natural disaster. But where should you store your backups? S3 allows you to store any data in the form of objects. The AWS object store is a perfect fit for your backup, allowing you to choose a location for your data as well as storing any amount of data with a pay-per-use pricing model.

In this section, you'll learn how to use the AWS CLI to upload data to and download data from S3. This approach isn't limited to off-site backups; you can use it in many other scenarios as well:

- Sharing files with your coworkers or partners, especially when working from different locations.
- Storing and retrieving artifacts needed to provision your virtual machines (such as application binaries, libraries, configuration files, and so on).
- Outsourcing storage capacity to lighten the burden on local storage systems—in particular, for data that is accessed infrequently.

First you need to create a bucket for your data on S3. As we mentioned earlier, the name of the bucket must be unique among all other S3 buckets, even those in other regions and those of other AWS customers. To find a unique bucket name, it's useful to use a prefix or suffix that includes your company's name or your own name. Run the following command in the terminal, replacing *$yourname* with your name:

```
$ aws s3 mb s3://awsinaction-$yourname
```

Your command should look similar to this one.

```
$ aws s3 mb s3://awsinaction-awittig
```

If your bucket name conflicts with an existing bucket, you'll get an error like this one:

```
[... ] An error occurred (BucketAlreadyExists) [...]
```

In this case, you'll need to use a different value for `$yourname`.

Everything is ready to upload your data. Choose a folder you'd like to back up, such as your Desktop folder. Try to choose a folder with a total size less than 1 GB, to avoid long waiting times and exceeding the Free Tier. The following command uploads the data from your local folder to your S3 bucket. Replace *`$Path`* with the path to your folder and *`$yourname`* with your name. `sync` compares your folder with the /backup folder in your S3 bucket and uploads only new or changed files:

```
$ aws s3 sync $Path s3://awsinaction-$yourname/backup
```

Your command should look similar to this one.

```
$ aws s3 sync /Users/andreas/Desktop s3://awsinaction-awittig/backup
```

Depending on the size of your folder and the speed of your internet connection, the upload can take some time.

After uploading your folder to your S3 bucket to back it up, you can test the restore process. Execute the following command in your terminal, replacing *`$Path`* with a folder you'd like to use for the restore (don't use the folder you backed up) and *`$yourname`* with your name. Your Downloads folder would be a good place to test the restore process:

```
$ aws s3 cp --recursive s3://awsinaction-$yourname/backup $Path
```

Your command should look similar to this one:

```
$ aws s3 cp --recursive s3://awsinaction-awittig/backup/ \
➥ /Users/andreas/Downloads/restore
```

Again, depending on the size of your folder and the speed of your internet connection, the download may take a while.

Versioning for objects

By default, S3 versioning is disabled for every bucket. Suppose you use the following steps to upload two objects:

1. Add an object with key *A* and data *1*.
2. Add an object with key *A* and data *2*.

(continued)

If you download (or *get*) the object with key *A*, you'll download data *2*. The old data *1* doesn't exist any more.

You can change this behavior by turning on *versioning* for a bucket. The following command activates versioning for your bucket. Don't forget to replace `$yourname`:

```
$ aws s3api put-bucket-versioning --bucket awsinaction-$yourname \
➥ --versioning-configuration Status=Enabled
```

If you repeat the previous steps, the first version of object *A* consisting of data *1* will be accessible even after you add an object with key *A* and data *2*. The following command retrieves all objects and versions:

```
$ aws s3api list-object-versions --bucket awsinaction-$yourname
```

You can now download all versions of an object.

Versioning can be useful for backup and archiving scenarios. Keep in mind that the size of the bucket you'll have to pay for will grow with every new version.

You no longer need to worry about losing data. S3 is designed for 99.999999999% durability of objects over a year. For instance, when storing 100,000,000,000 objects on S3, you will lose only a single object per year on average.

After you've successfully restored your data from the S3 bucket, it's time to clean up. Execute the following command to remove the S3 bucket containing all the objects from your backup. You'll have to replace `$yourname` with your name to select the right bucket. `rb` removes the bucket; the `force` option deletes every object in the bucket before the bucket itself is deleted:

```
$ aws s3 rb --force s3://awsinaction-$yourname
```

Your command should look similar to this one:

```
$ aws s3 rb --force s3://awsinaction-awittig
```

You're finished—you've uploaded and downloaded files to S3 with the help of the CLI.

Removing bucket causes BucketNotEmpty error

If you turn on versioning for your bucket, removing the bucket will cause a `BucketNotEmpty` error. Use the Management Console to delete the bucket in this case:

1. Open the Management Console with your browser.
2. Go to the S3 service using the main navigation menu.
3. Select your bucket.
4. Press the Delete bucket button and confirm your action in the dialog that opens.

8.4 Archiving objects to optimize costs

You used S3 to back up your data in the previous section. If you want to reduce the cost of backup storage, you should consider another AWS service: *Amazon Glacier*. The price of storing data with Glacier is about a fifth of what you pay to store data with S3. So what's the catch? S3 offers instant retrieval of your data. In contrast, you have to request your data and wait between one minute and twelve hours before your data is available when working with Glacier. Table 8.1 shows the differences between S3 and Glacier.

Table 8.1 Differences between storing data with S3 and Glacier

	S3	Glacier
Storage costs for 1 GB per month in US East (N. Virginia)	0.023 USD	0.004 USD
Costs for inserting data	Low	High
Costs for retrieving data	Low	High
Accessibility	Immediate upon request	One minute to twelve hours after request. Faster retrieval is more expensive.
Durability	Designed for annual durability of 99.999999999%	Designed for annual durability of 99.999999999%

Amazon Glacier is designed for archiving large files that you upload once and download seldom. It is expensive to upload and retrieve a lot of small files, so you should bundle small files into large archives before storing them on Amazon Glacier. You can use Glacier as a standalone service accessible via HTTPS, integrated into your backup solution, or use S3 integration as in the following example.

8.4.1 Creating an S3 bucket for the use with Glacier

In this section, you'll learn how to use Glacier to archive objects that have been stored on S3 to reduce storage costs. As a rule, only move data to Glacier if the chance you'll need to access the data later is low.

For example, suppose you are storing measurement data from temperature sensors on S3. The raw data is uploaded to S3 constantly and processed once a day. After the raw data has been analyzed, results are stored within a database. The raw data on S3 is no longer needed, but should be archived in case you need to re-run the data processing again in the future. Therefore, you move the raw measurement data to Glacier after one day to minimize storage costs.

The following example guides you through storing objects on S3, moving objects to Glacier, and restoring objects from Glacier. As illustrated in figure 8.4, you need to create a new S3 bucket:

1. Open the Management Console at https://console.aws.amazon.com.
2. Move to the S3 service using the main menu.

3 Click the Create button.
4 Type in a unique name for your bucket (such as awsincation-glacier-$yourname).
5 Choose US East (N. Virginia) as the region for the bucket.
6 Click the Next button.
7 Click the Create button on the last page of the wizard.

Figure 8.4 Creating an S3 bucket via the Management Console

8.4.2 *Adding a lifecycle rule to a bucket*

Back to our example: you are storing raw measurement data in an S3 bucket. The raw data has been analyzed. Next, the raw data should be archived on Glacier. To do so,

add a *lifecycle rule* to your bucket. A lifecycle rule can be used to *archive* or *delete* objects after a given number of days. To add a lifecycle rule that moves objects to Glacier, follow these steps, also illustrated in figure 8.5:

1. Select your bucket named *awsincation-glacier-$yourname* from the bucket overview.
2. Switch to the Management tab.
3. Click the Add Lifecycle Rule button.

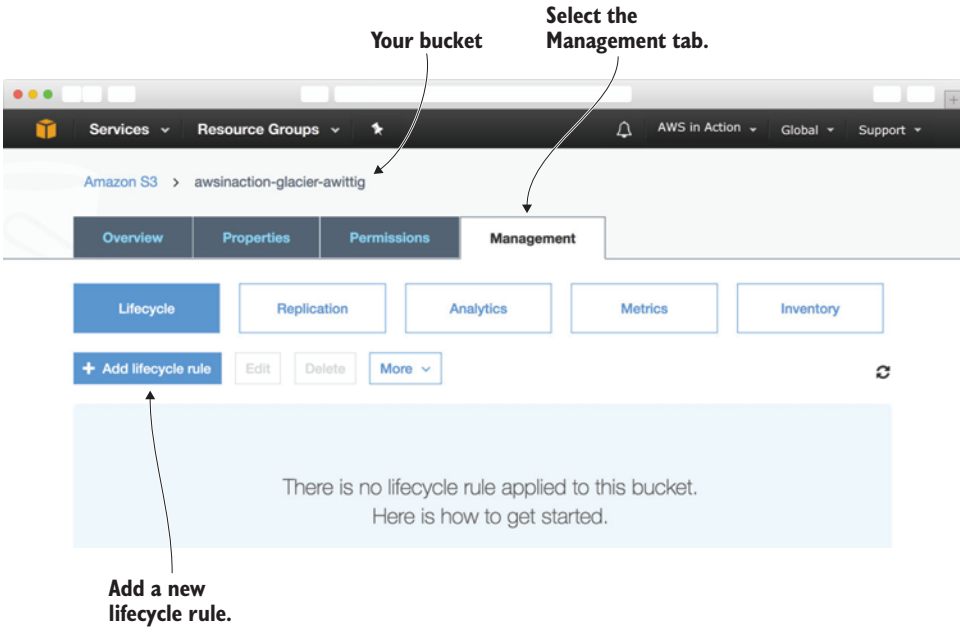


Figure 8.5 Adding a lifecycle rule to move objects to Glacier automatically

A wizard starts that will guide you through the process of creating a new lifecycle rule, as shown in figure 8.6. In the first step of the wizard, you are asked to provide a name and the scope for the lifecycle rule. Type in `glacier` as the rule name, and keep the filter that limits the scope of the rule empty: this applies the rule to all objects within your bucket.

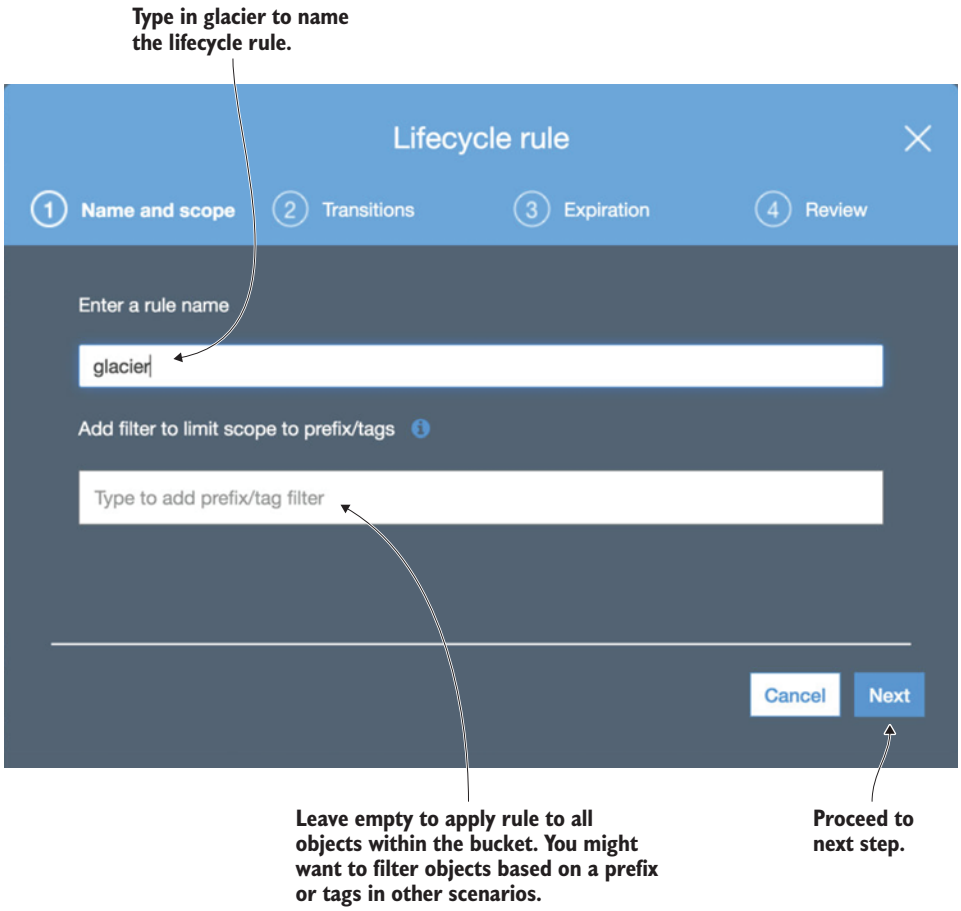


Figure 8.6 Choosing the name and scope of your lifecycle rule

In the next step of the wizard, you will configure the lifecycle rule to archive objects to Glacier. Figure 8.7 shows the details of configuring the transition.

1. Enable transitions for the current version of your objects. As you haven't enabled versioning for your bucket, previous versions of your objects are not available.
2. Select Transition to Amazon Glacier after as the transition type.
3. In the Days After Object Creation field, type in 0 to move your objects to Glacier as quickly as possible.
4. Click Next to proceed to the next step.

Skip the next step of the wizard, which allows you to configure a lifecycle rule to delete objects after a specified period of time. The last step of the wizard shows a summary of your lifecycle rule. Click Save to create your lifecycle rule.

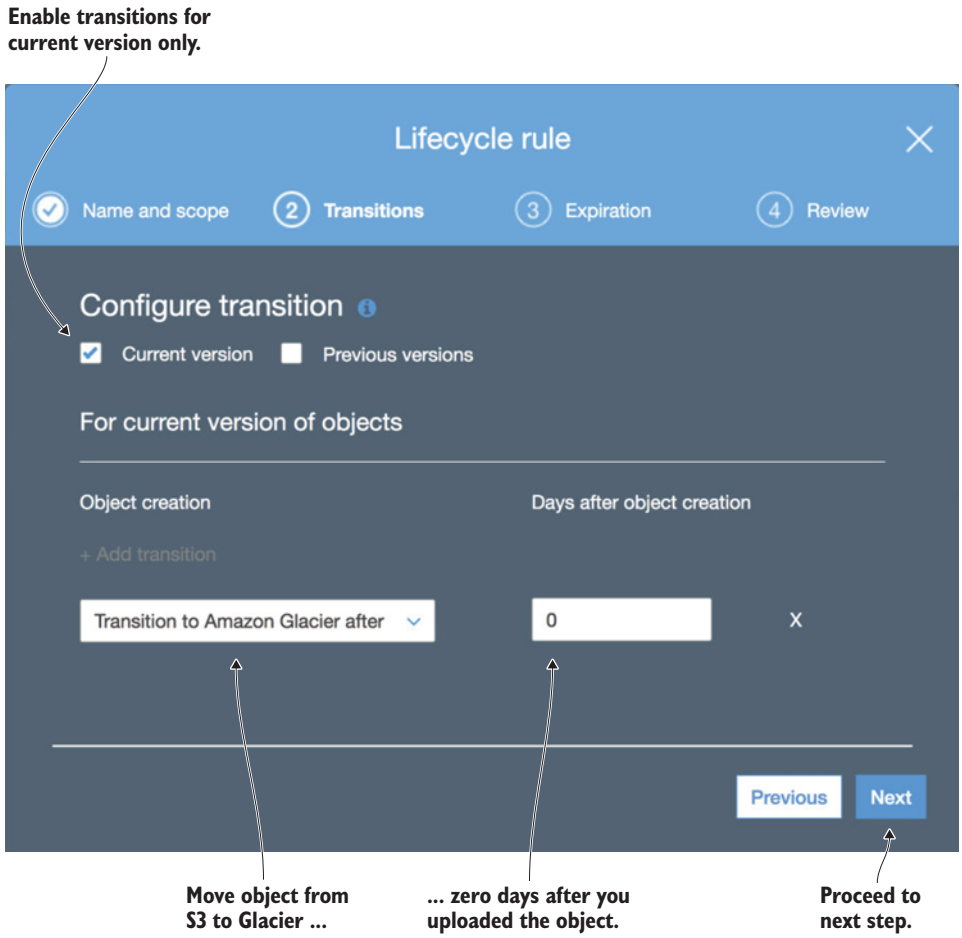


Figure 8.7 Enable transition to Amazon Glacier after 0 days

8.4.3 Experimenting with Glacier and your lifecycle rule

You've successfully created a lifecycle rule that will automatically move all objects from the bucket to Glacier.

> **NOTE** The following example will take more time than usual. It will take up to 24 hours for the lifecycle rule to move your objects to Glacier. The restore process from Glacier to S3 will take 3 to 5 hours.

It's now time to test the process of archiving your raw measurement data. Go back to the overview of your bucket named *awsincation-glacier-$yourname*. Upload a bunch of files by clicking the Upload button. As you probably don't have any files that include measurement data from temperature sensors at hand, feel free to use any kind of data. Your bucket will look similar to what is shown in figure 8.8. By default, all files are stored with storage class Standard, which means they're stored on S3.

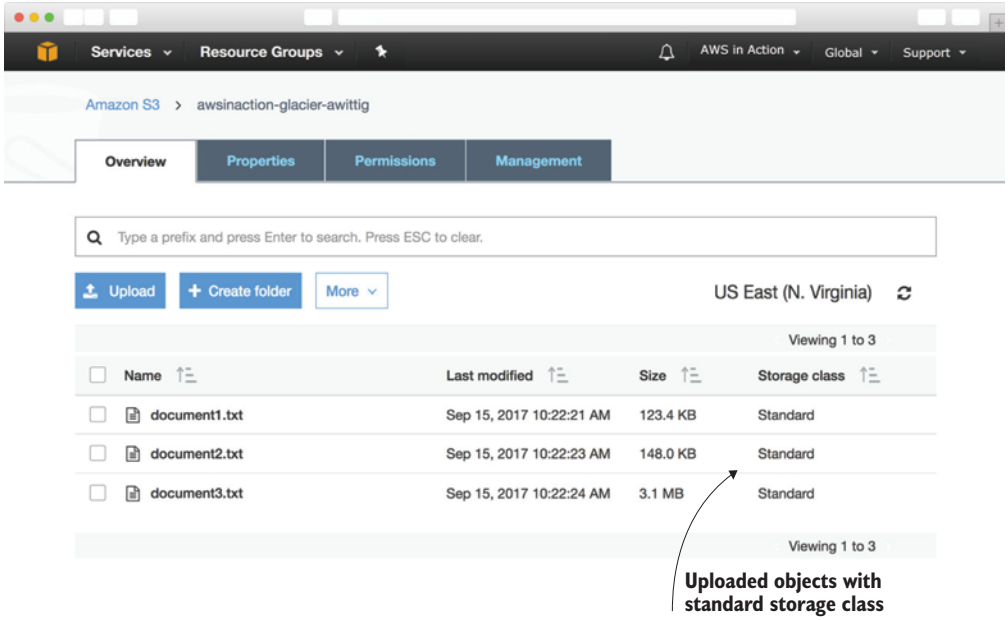


Figure 8.8 Objects with storage class Standard immediately after upload

The lifecycle rule will move the created objects to Glacier. But even though the chosen time gap is 0 days, the move will take up to 24 hours. After your objects have moved to Glacier, the storage class will switch to Glacier, as shown in figure 8.9.

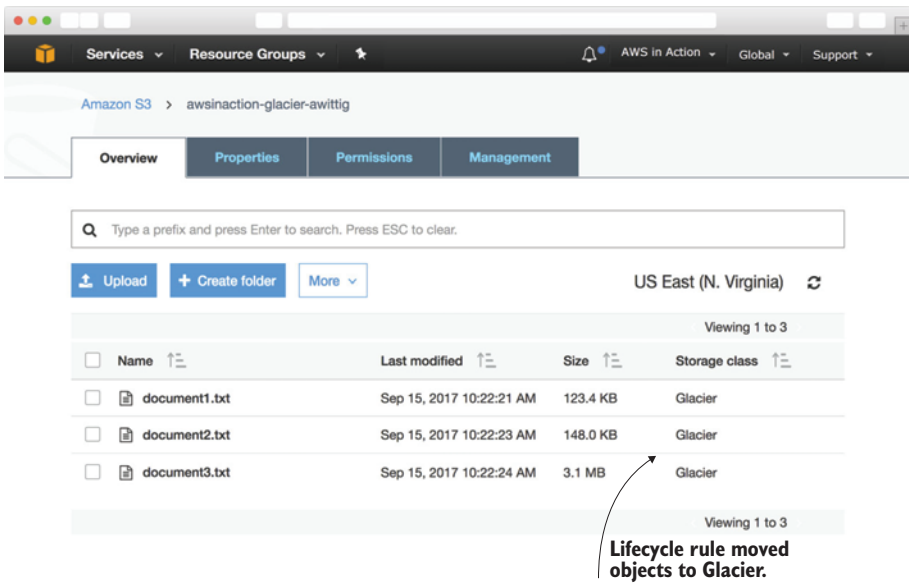


Figure 8.9 Objects have been moved to Glacier after 24 hours.

Let's assume you have found a bug in the processing of measurement data. Incorrect data has been extracted from the raw data and stored in your database. In order to re-run the data analytics, you have to restore the raw data from Glacier.

It is not possible to download files stored in Glacier directly, but you can restore an object from Glacier to S3. Follow the steps illustrated in figure 8.10 to trigger a restore using the Management Console:

1. Open the S3 bucket named *awsincation-glacier-$yourname*.
2. Select the object you want to restore from Glacier to S3.
3. Choose the Initiate Restore action, which is hidden under More.
4. A dialog appears in which you choose how many days the object will be available via S3 after the restore from Glacier, as well as the retrieval option.
5. Click OK to initiate the restore.

Figure 8.10 Restoring an object from Glacier to S3

Retrieval options

Amazon Glacier offers three retrieval options:

- *Expedited*. Data is available within 1–5 minutes; this is the most expensive restore option.
- *Standard*. Data is available within 3–5 hours, for a modest charge.
- *Bulk*. Data is available within 5–12 hours; this is the least expensive restore option.

Restoring an object with the Standard retrieval option usually takes 3 to 5 hours. After the restore is complete, you can download the object. You could re-run your data processing on the raw data now.

Cleaning up

Delete your bucket after you finish the Glacier example. You can do this with the help of the Management Console by following theses steps:

1. Go to the overview of S3 buckets.
2. Select the bucket named *awsincation-glacier-$yourname*.
3. Click the Delete bucket button and confirm the action within the shown dialog.

You've learned how to use S3 with the help of the CLI and the Management Console. We'll show you how to integrate S3 into your applications with the help of SDKs in the next section.

8.5 Storing objects programmatically

S3 is accessible using an API via HTTPS. This enables you to integrate S3 into your applications by making requests to the API programmatically. Doing so allows your applications to benefit from a scalable and highly available data store. AWS offers free SDKs for common programming languages like Go, Java, JavaScript, PHP, Python, Ruby, and .NET. You can execute the following operations using an SDK directly from your application:

- Listing buckets and their objects.
- Creating, removing, updating, and deleting (CRUD) objects and buckets.
- Managing access to objects.

Here are examples of how you can integrate S3 into your application:

- *Allow a user to upload a profile picture.* Store the image in S3, and make it publicly accessible. Integrate the image into your website via HTTPS.
- *Generate monthly reports (such as PDFs), and make them accessible to users.* Create the documents and upload them to S3. If users want to download documents, fetch them from S3.

- *Share data between applications.* You can access documents from different applications. For example, application A can write an object with the latest information about sales, and application B can download the document and analyze the data.

Integrating S3 into an application is one way to implement the concept of a *stateless server*. We'll show you how to integrate S3 into your application by diving into a simple web application called Simple S3 Gallery next. This web application is built on top of Node.js and uses the AWS SDK for JavaScript and Node.js. You can easily transfer what you learn from this example to SDKs for other programming languages; the concepts are the same.

> **Installing and getting started with Node.js**
>
> Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.
>
> After Node.js is installed, you can verify that everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v8.*`. Now you're ready to run JavaScript examples like the Simple S3 Gallery.
>
> Do you want to get started with Node.js? We recommend *Node.js in Action (2nd edition)* from Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* from P.J. Evans, (Manning, 2018).

Simple S3 Gallery allows you to upload images to S3 and displays all the images you've already uploaded. Figure 8.11 shows Simple S3 Gallery in action. Let's set up S3 to start your own gallery.

8.5.1 Setting up an S3 bucket

To begin, you need to set up an empty bucket. Execute the following command, replacing *$yourname* with your name or nickname:

```
$ aws s3 mb s3://awsinaction-sdk-$yourname
```

Your bucket is now ready to go. Installing the web application is the next step.

8.5.2 Installing a web application that uses S3

You can find the Simple S3 Gallery application in /chapter08/gallery/ in the book's code folder. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

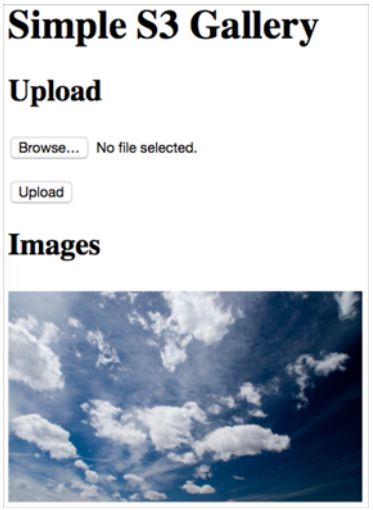


Figure 8.11 The Simple S3 Gallery app lets you upload images to an S3 bucket and then download them from the bucket for display.

To start the web application, run the following command. Replace `$yourname` with your name; the name of the S3 bucket is then passed to the web application:

```
$ node server.js awsinaction-sdk-$yourname
```

> **Where is the code located?**
> You can find all the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip.

After you start the server, you can open the gallery application. To do so, open http://localhost:8080 with your browser. Try uploading a few images.

8.5.3 Reviewing code access S3 with SDK

You've uploaded images to Simple S3 Gallery and displayed images from S3. Inspecting parts of the code will help you to understand how you can integrate S3 into your own applications. It's not a problem if you don't follow all the details of the programming language (JavaScript) and the Node.js platform; we just want you to get an idea of how to use S3 via SDKs.

UPLOADING AN IMAGE TO S3

You can upload an image to S3 with the SDK's `putObject()` function. Your application will connect to the S3 service and transfer the image via HTTPS. The following listing shows how to do so.

Listing 8.1 Uploading an image with the AWS SDK for S3

```
var AWS = require('aws-sdk');
var uuid = require('uuid');

var s3 = new AWS.S3({'region': 'us-east-1'});

var bucket = [...];

function uploadImage(image, response) {
  var params = {
    Body: image,
    Bucket: bucket,
    Key: uuid.v4(),
    ACL: 'public-read',
    ContentLength: image.byteCount,
    ContentType: image.headers['content-type']
  };
  s3.putObject(params, function(err, data) {
    if (err) {
      console.error(err);
      response.status(500);
```

```
      response.send('Internal server error.');
    } else {
      response.redirect('/');
    }
  });
}
```

Handles success

The AWS SDK takes care of sending all the necessary HTTPS requests to the S3 API in the background.

Listing all the images in the S3 bucket

To display a list of images, the application needs to list all the objects in your bucket. This can be done with the S3 service's `listObjects()` function. The next listing shows the implementation of the corresponding function in the server.js JavaScript file, acting as a web server.

Listing 8.2 Retrieving all the image locations from the S3 bucket

```
var bucket = [...];

function listImages(response) {
  var params = {
    Bucket: bucket
  };
  s3.listObjects(params, function(err, data) {
    if (err) {
      console.error(err);
      response.status(500);
      response.send('Internal server error.');
    } else {
      var stream = mu.compileAndRender(
        'index.html',
        {
          Objects: data.Contents,
          Bucket: bucket
        }
      );
      stream.pipe(response);
    }
  });
}
```

Defines parameters for the list-objects operation

Calls the list-objects operation

The resulting data contains the objects from the bucket list.

Listing the objects returns the names of all the images from the bucket, but the list doesn't include the image content. During the uploading process, the access rights to the images are set to public read. This means anyone can download the images with the bucket name and a random key. The following listing shows an excerpt of the index.html template, which is rendered on request. The `Objects` variable contains all the objects from the bucket.

Listing 8.3 Template to render the data as HTML

Iterates over all objects

Puts together the URL to fetch an image from the bucket

```
[...]
<h2>Images</h2>
{{#Objects}}
  <p><img src=
➥ "https://s3.amazonaws.com/{{Bucket}}/{{Key}}"
➥ width="400px" ></p>
{{/Objects}}
[...]
```

You've now seen the three important parts of the Simple S3 Gallery integration with S3: uploading an image, listing all images, and downloading an image.

Cleaning up

Don't forget to clean up and delete the S3 bucket used in the example. Use the following command, replacing *$yourname* with your name:

```
$ aws s3 rb --force s3://awsinaction-sdk-$yourname
```

You've learned how to use S3 using the AWS SDK for JavaScript and Node.js. Using the AWS SDK for other programming languages is similar.

8.6 Using S3 for static web hosting

We have started our blog cloudonaut.io in May 2015. The most popular blog posts like "5 AWS mistakes you should avoid" (https://cloudonaut.io/5-aws-mistakes-you-should-avoid/), "Integrate SQS and Lambda: serverless architecture for asynchronous workloads" (http://mng.bz/8m5n), and "AWS Security Primer" (https://cloudonaut.io/aws-security-primer/) have been read more than 165,000 times. But we didn't need to operate any VMs to publish our blog posts. Instead we used S3 to host our static website built with a static site generator https://hexo.io. This approach provides a cost effective, scalable, and maintenance-free infrastructure for our blog.

You can host a static website with S3 and deliver static content like HTML, JavaScript, CSS, images (such as PNG and JPG), audio, and videos as well. But keep in mind that you can't execute server-side scripts like PHP or JSP. For example, it's not possible to host WordPress, a CMS based on PHP, on S3.

Increasing speed by using a CDN

Using a content-delivery network (CDN) helps reduce the load time for static web content. A CDN distributes static content like HTML, CSS, and images to nodes all around the world. If a user sends out a request for some static content, the request is answered from the nearest available node with the lowest latency.

> Various providers offer CDNs. Amazon CloudFront is the CDN offered by AWS. When using CloudFront, users connect to CloudFront to access your content, which is which is fetched from S3 or other sources. See the CloudFront documentation at http://mng.bz/Kctu if you want to set this up; we won't cover it in this book.

In addition, S3 offers the following features for hosting a static website:

- *Defining a custom index document and error documents.* For example, you can define index.html as the default index document.
- *Defining redirects for all or specific requests.* For example, you can forward all requests from /img/old.png to /img/new.png.
- *Setting up a custom domain for S3 bucket.* For example, Andreas might want to set up a domain like mybucket.andreaswittig.info for my bucket.

8.6.1 Creating a bucket and uploading a static website

First you need to create a new S3 bucket. To do so, open your terminal and execute the following command, replacing `$BucketName` with your own bucket name. (As we've mentioned, the bucket name has to be globally unique. If you want to redirect your domain name to S3, you must use your entire domain name as the bucket name.)

```
$ aws s3 mb s3://$BucketName
```

The bucket is empty; you'll place an HTML document in it next. We've prepared a placeholder HTML file. Download it to your local machine from the following URL: http://mng.bz/8ZPS. You can now upload the file to S3. Execute the following command to do so, replacing `$PathToPlacerholder` with the path to the HTML file you downloaded in the previous step and `$BucketName` with the name of your bucket:

```
$ aws s3 cp $PathToPlaceholder/helloworld.html \
➥ s3://$BucketName/helloworld.html
```

You've now created a bucket and uploaded an HTML document called helloworld .html. You need to configure the bucket next.

8.6.2 Configuring a bucket for static web hosting

By default, only you, the owner, can access files from your S3 bucket. You want to use S3 to deliver your static website, so you'll need to allow everyone to view or download the documents included in your bucket. A *bucket policy* helps you control access to bucket objects globally. You already know from chapter 6 that policies are defined in JSON and contain one or more statements that either allow or deny specific actions on specific resources. Bucket policies are similar to IAM policies.

Download our bucket policy from the following URL: http://mng.bz/HhgR. You need to edit the bucketpolicy.json file next, as shown in the following listing. Open the file with the editor of your choice, and replace *$BucketName* with the name of your bucket.

Listing 8.4 Bucket policy allowing read-only access to every object in a bucket

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Sid":"AddPerm",
      "Effect":"Allow",                          <-- Allows access
      "Principal": "*",                          <-- For anyone
      "Action":["s3:GetObject"],                 <-- Read objects
      "Resource":["arn:aws:s3:::$BucketName/*"]  <-- Your bucket
    }
  ]
}
```

You can add a bucket policy to your bucket with the following command. Replace *$BucketName* with the name of your bucket and *$PathToPolicy* with the path to the bucketpolicy.json file:

```
$ aws s3api put-bucket-policy --bucket $BucketName \
➥ --policy file://$PathToPolicy/bucketpolicy.json
```

Every object in the bucket can now be downloaded by anyone. You need to enable and configure the static web-hosting feature of S3 next. To do so, execute the following command, replacing *$BucketName* with the name of your bucket:

```
$ aws s3 website s3://$BucketName --index-document helloworld.html
```

Your bucket is now configured to deliver a static website. The HTML document helloworld.html is used as index page. You'll learn how to access your website next.

8.6.3 Accessing a website hosted on S3

You can now access your static website with a browser. To do so, you need to choose the right endpoint. The endpoints for S3 static web hosting depend on your bucket's region:

```
http://$BucketName.s3-website-$Region.amazonaws.com
```

Replace *$BucketName* with your bucket name and *$Region* with your region. So if your bucket is called `AwesomeBucket` and was created in the default region us-east-1, your bucket name would be:

```
http://AwesomeBucket.s3-website-us-east-1.amazonaws.com
```

Open this URL with your browser, and you should be welcomed by a Hello World website.

Linking a custom domain to an S3 bucket

If you want to avoid hosting static content under a domain like awsinaction.s3-website-us-east-1.amazonaws.com, you can link a custom domain to an S3 bucket, such as `awsinaction.example.com`. All you have to do is to add a CNAME record for your domain, pointing to the bucket's S3 endpoint.

The CNAME record will only work if you comply with the following rules:

- *Your bucket name must match the CNAME record name*. For example, if you want to create a CNAME for `awsinaction.example.com`, your bucket name must be `awsinaction.example.com` as well.
- *CNAME records won't work for the primary domain name (such as `example.com`)*. You need to use a subdomain for CNAMEs like *awsinaction* or *www*, for example. If you want to link a primary domain name to an S3 bucket, you need to use the Route 53 DNS service from AWS.

Linking a custom domain to your S3 bucket only works for HTTP. If you want to use HTTPS (and you probably should), use AWS CloudFront together with S3. AWS CloudFront accepts HTTPS from the client and forwards the request to S3.

Cleaning up

Don't forget to clean up your bucket after you finish the example. To do so, execute the following command, replacing `$BucketName` with the name of your bucket:

```
$ aws s3 rb --force s3://$BucketName
```

8.7 Best practices for using S3

If you're using S3 via the CLI or integrating it into your applications, it's valuable to know about how the object store works. One big difference between S3 and many storage solutions is the fact that S3 is *eventually consistent*, which means you might read stale data after changing an object for a short period of time. Usually you will read the latest version of an object within less than a second after a write, but in rare cases, you might read stale data for much longer. If you don't consider this, you may be surprised if you try to read objects immediately after changing them. Another challenge is designing object keys that offer maximum I/O performance on S3. You'll learn more about both topics next.

8.7.1 Ensuring data consistency

If you're creating, updating, or deleting an object on S3, this operation is *atomic*. This means that if you're reading an object after a create, an update, or a delete, you'll never get corrupted or partial data. But it's possible that a read could return the old data for a while.

If you create, update or delete an object and your request is successful, your change is safely stored. But accessing the changed object immediately might return the *old* version, as shown in figure 8.12. If you retry accessing the object, after a while the new version will be available.

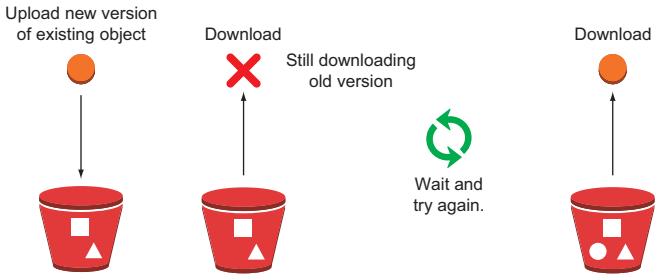


Figure 8.12 Eventual consistency: if you update an object and try to read it, the object may contain the old version. After some time passes, the latest version will be available.

If you don't send a GET or HEAD request to the key of an object before creating the object with a PUT request, S3 offers read-after-write consistency in all regions.

8.7.2 Choosing the right keys

Naming variables and files is difficult. This is especially true for choosing the right keys for objects you want to store in S3. In S3, keys are stored in alphabetical order in an index. The key name determines which partition the key is stored in. If your keys all begin with the same characters, this will limit the I/O performance of your S3 bucket. If your workload will require more than 100 requests per second, you should choose keys for your objects that begin with different characters. As figure 8.13 shows, this will give you maximum I/O performance.

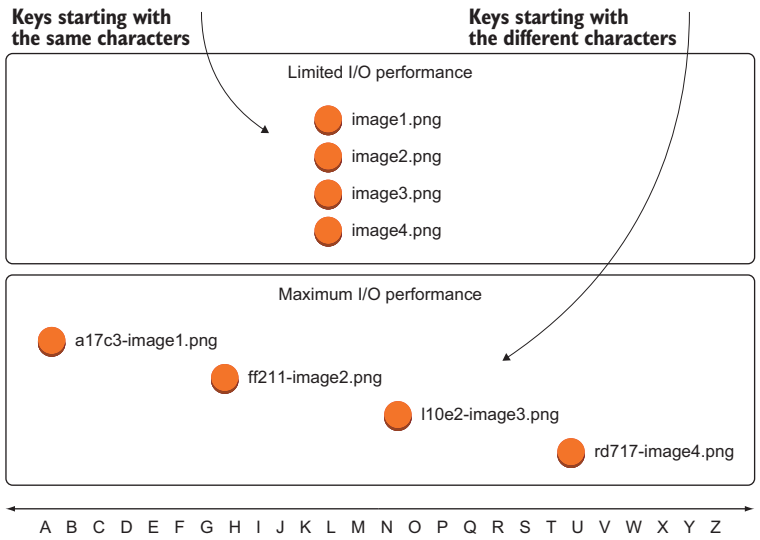


Figure 8.13 To improve I/O performance with S3, don't use keys that start with the same characters.

Using a slash (/) in the key name acts like creating a folder for your object. If you create an object with the key *folder/object.png*, the folder will become visible as *folder* if you're browsing your bucket with a GUI like the Management Console, for example. But technically, the key of the object still is folder/object.png.

Suppose you need to store images that were uploaded by different users. You might come up with the following naming schema for your object keys:

```
upload/images/$ImageId.png
```

$ImageId is a numerical ID that is increased with each new image. A list of your objects might look like this.

```
image1.png
image2.png
image3.png
image4.png
```

The object keys are in alphabetical order, and your maximum throughput with S3 won't be optimal. You can fix this by adding a hash prefix to each object. For example, you can use the MD5 hash of the original key name and prepend it to the key:

```
a17c3-image1.png
ff211-image2.png
l10e2-image3.png
rd717-image4.png
```

This will help distribute your keys across partitions and increase the I/O performance of S3. Knowing about the internals of S3 helps you to optimize your usage.

Summary

- An object consists of a unique identifier, metadata to describe and manage the object, and the content itself. You can save images, documents, executables, or any other content as an object in an object store.
- Amazon S3 is an object store accessible only via HTTP(S). You can upload, manage, and download objects with the CLI, SDKs, or the Management Console.
- Integrating S3 into your applications will help you implement the concept of a stateless server, because you don't have to store objects locally on the server.
- You can define a lifecycle for your objects that will move them from Amazon S3 to Amazon Glacier, a special service for archiving data that you don't need to access frequently. Doing so reduces your cost dramatically.
- S3 is an eventually consistent object store. You have to consider this if you integrate it into your applications and processes, to avoid unpleasant surprises.

9 Storing data on hard drives: EBS and instance store

This chapter covers

- Attaching persistent storage volumes to EC2 instance
- Using temporary storage attached to the host system
- Backing up volumes
- Testing and tweaking volume performance
- Differences between persistent (EBS) and temporary volumes (instance store)

Imagine your task is to migrate an enterprise application from being hosted on-premises to AWS. Typically, legacy applications read and write files from a file system. Switching to object storage, as described in the previous chapter, is therefore not possible. Fortunately, AWS offers good old block-level storage as well, allowing you to migrate your legacy application without the need for expensive modifications.

Block-level storage with a disk file system (FAT32, NTFS, ext3, ext4, XFS, and so on) can be used to store files as you would on a personal computer. A *block* is a sequence of bytes, and the smallest addressable unit. The OS is the intermediary between the application that wants to access files and the underlying file system

and block-level storage. The disk file system manages where (at what block address) your files are stored. You can use block-level storage only in combination with an EC2 instance where the OS is running.

The OS provides access to block-level storage via open, write, and read system calls. The simplified flow of a read request goes like this:

1. An application wants to read the file /path/to/file.txt and makes a read system call.
2. The OS forwards the read request to the file system.
3. The file system translates /path/to/file.txt to the block on the disk where the data is stored.

Applications like databases that read or write files by using system calls must have access to block-level storage for persistence. You can't tell a MySQL database to store its files in an object store because MySQL uses system calls to access files.

> **Not all examples are covered by the Free Tier**
>
> The examples in this chapter are *not* all covered by the Free Tier. A special warning message appears when an example incurs costs. Nevertheless, as long as you don't run all other examples longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

AWS provides two kinds of block-level storage:

- A *persistent block-level storage* volume connected via network—This is the best choice for most problems, because it is independent from your virtual machine's life cycle and replicates data among multiple disks automatically to increase durability and availability.
- A *temporary block-level storage* volume physically attached to the host system of the virtual machine—This is interesting if you're optimizing for performance, as it is directly attached to the host system and therefore offers low latency and high throughput when accessing your data.

The next three sections will introduce and compare these two solutions by connecting storage with an EC2 instance, doing performance tests, and exploring how to back up the data.

9.1 Elastic Block Store (EBS): Persistent block-level storage attached over the network

Elastic Block Store (EBS) provides persistent block-level storage with built-in data replication. Typically EBS is used in the following scenarios:

- Operating a relational database system on a virtual machine.
- Running a (legacy) application that requires a filesystem to store data on EC2.
- Storing and booting the operating system of a virtual machine.

An EBS volume is separate from an EC2 instance and connected over the network, as shown in figure 9.1. EBS volumes:

- Aren't part of your EC2 instances; they're attached to your EC2 instance via a network connection. If you terminate your EC2 instance, the EBS volumes remain.
- Are either not attached to an EC2 instance or attached to exactly one EC2 instance at a time.
- Can be used like typical hard disks.
- Replicate your data on multiple disks to prevent data loss due to hardware failures.

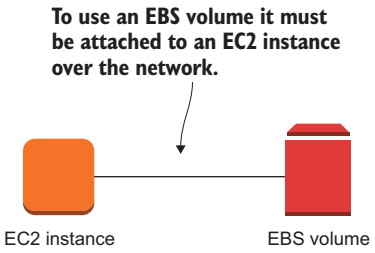


Figure 9.1 EBS volumes are independent resources but can only be used when attached to an EC2 instance.

EBS volumes have one big advantage: they are not part of the EC2 instance; they are an independent resource. No matter if you stop your virtual machine or your virtual machine fails because of a hardware defect, your volume and your data will remain.

WARNING You can't attach the same EBS volume to multiple virtual machines! See chapter 10 if you are looking for a network filesystem.

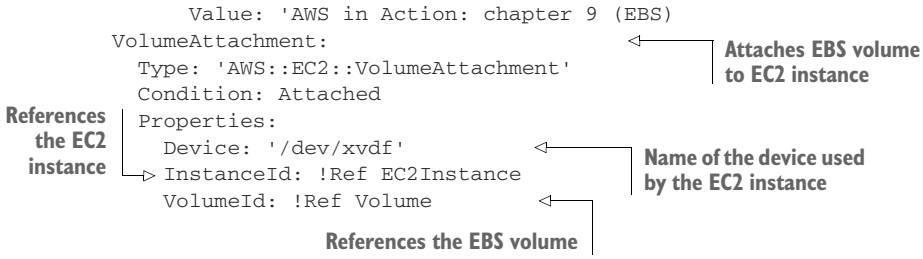
9.1.1 Creating an EBS volume and attaching it to your EC2 instance

Let's return to the example from the beginning of the chapter. You are migrating a legacy application to AWS. The application needs to access a filesystem to store data. As the data contains business-critical information, durability and availability are important. Therefore, you create an EBS volume for persistent block storage. The legacy application runs on a virtual machine, and the volume is attached to the VM to enable access to the block-level storage.

The following bit of code demonstrates how to create an EBS volume and attach it to an EC2 instance with the help of CloudFormation:

```
EC2Instance:                                    <-- Defines the EC2 instance
  Type: 'AWS::EC2::Instance'
  Properties:                                   <-- We are skipping the properties of
    # [...]                                         the EC2 instance in this example.

Volume:                                         <-- Defines the EBS volume
  Type: 'AWS::EC2::Volume'
  Properties:
    AvailabilityZone: !Sub ${EC2Instance.AvailabilityZone}
    Size: 5                                     <-- Creates a volume with 5 GB capacity
    VolumeType: gp2                             <-- Default volume type based on SSD
    Tags:
    - Key: Name
```



An EBS volume is a standalone resource. This means your EBS volume can exist without an EC2 instance, but you need an EC2 instance to access the EBS volume.

9.1.2 Using EBS

To help you explore EBS, we've prepared a CloudFormation template located at http://mng.bz/6q51. Create a stack based on that template by clicking the CloudFormation Quick-Create link at http://mng.bz/5o2D, select the default VPC and a random subnet, and set the `AttachVolume` parameter to `yes`. Don't forget to check the box marked I acknowledge that AWS CloudFormation might create IAM resources. After creating the stack, copy the `PublicName` output and connect via SSH: `ssh -i` *$PathToKey*`/ mykey.pem ec2-user@`*$PublicName*.

You can see the attached EBS volumes using `fdisk`. Usually, EBS volumes can be found somewhere in the range of /dev/xvdf to /dev/xvdp. The root volume (/dev/xvda) is an exception—it's based on the AMI you choose when you launch the EC2 instance, and contains everything needed to boot the instance (your OS files):

```
$ sudo fdisk -l
Disk /dev/xvda: 8589 MB [...]
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: gpt


  #         Start          End     Size  Type            Name
  1          4096     16777182       8G  Linux filesyste Linux
128          2048         4095       1M  BIOS boot parti BIOS Boot Partition

Disk /dev/xvdf: 5368 MB [...]
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

The root volume, an EBS volume with a size of 8 GiB.

An additional volume, an EBS volume with a size of 5 GiB.

The first time you use a newly created EBS volume, you must create a filesystem. You could also create partitions, but in this case the volume size is only 5 GB, so you probably don't want to split it up further. As you can create EBS volumes in any size and attach multiple volumes to your VM, partitioning a single EBS volume is uncommon.

Instead, you should create volumes at the size you need; if you need two separate scopes, create two volumes. In Linux, you can create a filesystem on the additional volume with the help of `mkfs`. The following example creates an ext4 file system:

```
$ sudo mkfs -t ext4 /dev/xvdf
mke2fs 1.42.12 (29-Aug-2014)
Creating filesystem with 1310720 4k blocks and 327680 inodes
Filesystem UUID: e9c74e8b-6e10-4243-9756-047ceaf22abc
Superblock backups stored on blocks:
  32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

After the filesystem has been created, you can mount the device:

```
$ sudo mkdir /mnt/volume/
$ sudo mount /dev/xvdf /mnt/volume/
```

To see mounted volumes, use `df -h`:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        489M   60K  488M   1% /dev
tmpfs           497M     0  497M   0% /dev/shm
/dev/xvda1      7.8G  980M  6.7G  13% /           <-- Root volume
/dev/xvdf       4.8G   10M  4.6G   1% /mnt/volume  <-- Additional volume
```

EBS volumes are independent from your virtual machine. To see this in action, you will save a file to a volume, unmount, and detach the volume. Afterward, you will attach and mount the volume again. The data will still be available!

```
$ sudo touch /mnt/volume/testfile    <-- Creates testfile in /mnt/volume/
$ sudo umount /mnt/volume/           <-- Unmounts the volume
```

Update the CloudFormation stack, and change the `AttachVolume` parameter to `no`. This will detach the EBS volume from the EC2 instance. After the update is completed, only your root device is left:

```
$ sudo fdisk -l
Disk /dev/xvda: 8589 MB, 8589934592 bytes, 16777216 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

```
Disk label type: gpt


  #        Start          End    Size  Type            Name
  1         4096     16777182      8G  Linux filesyste Linux
128         2048         4095      1M  BIOS boot parti BIOS Boot Partition
```

The testfile in /mnt/volume/ is also gone:

```
$ ls /mnt/volume/testfile
ls: cannot access /mnt/volume/testfile: No such file or directory
```

Now you'll attach the EBS volume again. Update the CloudFormation stack, and change the `AttachVolume` parameter to `yes`. After the update is completed, /dev/xvdf is available again:

```
$ sudo mount /dev/xvdf /mnt/volume/
$ ls /mnt/volume/testfile
/mnt/volume/testfile
```

Mounts the attached volume again

Checks whether testfile is still in /mnt/volume/

Voilà: the file test file that you created in /mnt/volume/ is still there.

9.1.3 Tweaking performance

Performance testing of hard disks is divided into read and write tests. To test the performance of your volumes, you will use a simple tool named `dd`, which can perform block-level reads and writes between a source `if=/path/to/source` and a destination `of=/path/to/destination`:

Writes 1 MB 1,024 times

```
$ sudo dd if=/dev/zero of=/mnt/volume/tempfile bs=1M count=1024 \
➥ conv=fdatasync,notrunc
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 16.9858 s, 63.2 MB/s
```

63.2 MB/s write performance

```
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
```

Flushes caches

```
$ sudo dd if=/mnt/volume/tempfile of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 16.3157 s, 65.8 MB/s
```

Reads 1 MB 1,024 times

65.8 MB/s read performance

Keep in mind that performance can be different depending on your actual workload. This example assumes that the file size is 1 MB. If you're hosting websites, you'll most likely deal with lots of small files instead.

But EBS performance is more complicated. Performance depends on the type of EC2 instance as well as the EBS volume type. Table 9.1 gives an overview of EC2 instance types that are EBS-optimized by default as well as types that can be optimized for an additional hourly charge. EC2 instances with EBS optimization benefit by having dedicated bandwidth to their EBS volumes. Input/output operations per second (IOPS) are measured using a standard 16 KB I/O operation size. Performance depends heavily on your workload: read versus write, as well as the size of your I/O operations (a bigger operation size equates to more throughput). These numbers are illustrations, and your mileage may vary.

Table 9.1 What performance can be expected from EBS optimized instance types?

Use case	Instance type	Max bandwidth (MiB/s)	Max IOPS	EBS optimized by default?
General purpose (3rd generation)	m3.xlarge–m3.2xlarge	60-119	4,000-8,000	No
General purpose (4th generation)	m4.large–m4.16xlarge	54–1192	3,600–65,000	Yes
General purpose (5th generation)	m5.large–m5.24xlarge	57–1192	3,600–65,000	Yes
Compute optimized (3rd generation)	c3.xlarge–c3.4xlarge	60–238	4,000–16,000	No
Compute optimized (4th generation)	c4.large–c4.8xlarge	60–477	4,000–32,000	Yes
Compute optimized (5th generation)	c5.large–c5.18xlarge	63–1073	4,000–64,000	Yes
Memory optimized	r4.large–r4.16xlarge	51–1,669	3,000–75,000	Yes
Storage optimized	i3.large–i3.16xlarge	51–1,669	3,000–65,000	Yes
Storage optimized	d2.xlarge–d2.8xlarge	89–477	6,000–32,000	Yes

Depending on your storage workload, you must choose an EC2 instance that can deliver the bandwidth you require. Additionally, your EBS volume must be balanced with the amount of bandwidth. Table 9.2 shows the different EBS volume types and how they perform.

Table 9.2 How EBS volume types differ

Volume type	Volume size	MiB/s	IOPS	Performance burst	Price
General Purpose SSD (gp2)	1 GiB–16 TiB	160	3 per GiB (up to 10,000)	3,000 IOPS	$ $ $
Provisioned IOPS SSD (io1)	4 GiB–16 TiB	500	As much as you provision (up to 50 IOPS per GiB or 32,000 IOPS)	n/a	$ $ $ $

Table 9.2 How EBS volume types differ *(continued)*

Volume type	Volume size	MiB/s	IOPS	Performance burst	Price
Throughput Optimized HDD (st1)	500 GiB-16 TiB	40 per TiB (up to 500)	500	250 MiB/s per TiB (up to 500 MiB/s)	$ $
Cold HDD (sc1)	500 GiB-16 TiB	12 per TiB (up to 250)	250	80 MiB/s per TiB (up to 250 MiB/s)	$
EBS Magnetic HDD (standard)	1 GiB–1 TiB	40–90	40-200 (100 on average)	Hundreds	$ $

Here are typical scenarios for the different volume types:

- Use *General Purpose SSD (gp2)* as the default for most workloads with medium load and a random access pattern. For example, use this as the boot volume or for all kinds of applications with low-to-medium I/O load.
- I/O-intensive workloads access small amounts of data randomly. *Provisioned IOPS SSD (io1)* offers throughput guarantees, for example, for large and business-critical database workloads.
- Use *Throughput Optimized HDD (st1)* for workloads with sequential I/O and huge amounts of data, such as Big Data workloads. Don't use this volume type for workloads in need of small and random I/O.
- *Cold HDD (sc1)* is a good fit when you are looking for a low-cost storage option for data you need to access infrequently and sequentially. Don't use this volume type for workloads in need of small and random I/O.
- *EBS Magnetic HDD (standard)* is an older volume type from a previous generation. It might be a good option when you need to access your data infrequently.

Gibibyte (GiB) and Tebibyte (TiB)

The terms *gibibyte* (GiB) and *tebibyte* (TiB) aren't used often; you're probably more familiar with gigabyte and terabyte. But AWS uses them in some places. Here's what they mean:

- 1 TiB = 2^{40} bytes = 1,099,511,627,776 bytes
- 1 TiB is ~ 1.0995 TB
- 1 TB = 10^{12} bytes = 1,000,000,000,000 bytes
- 1 GiB = 2^{30} bytes = 1,073,741,824 bytes
- 1 GiB is ~ 1.074 GB
- 1 GB = 10^{9} bytes = 1,000,000,000 bytes

EBS volumes are charged based on the size of the volume, no matter how much data you store in the volume. If you provision a 100 GiB volume, you pay for 100 GiB even if you have no data on the volume. If you use EBS Magnetic HDD (standard) volumes, you must also pay for every I/O operation you perform. Provisioned IOPS SSD (io1)

volumes are additionally charged based on the provisioned IOPS. Use the AWS Simple Monthly Calculator at http://aws.amazon.com/calculator to determine how much your storage setup will cost.

We advise you to use general-purpose (SSD) volumes as the default. If your workload requires more IOPS, then go with provisioned IOPS (SSD). You can attach multiple EBS volumes to a single EC2 instance to increase overall capacity or for additional performance.

9.1.4 Backing up your data with EBS snapshots

EBS volumes replicate data on multiple disks automatically and are designed for an annual failure rate (AFR) of 0.1% and 0.2%. This means on average you should expect to lose 0.5–1 of 500 volumes per year. To plan for an unlikely (but possible) failure of an EBS volume, or more likely a human failure, you should create backups of your volumes regularly. Fortunately, EBS offers an optimized, easy-to-use way to back up EBS volumes with EBS snapshots. A *snapshot* is a block-level incremental backup that is stored in S3. If your volume is 5 GiB in size and you use 1 GiB of data, your first snapshot will be around 1 GiB in size. After the first snapshot is created, only the changes will be saved to S3, to reduce the size of the backup. EBS snapshots are charged based on how many gigabytes you use.

You'll now create a snapshot using the CLI. Before you can do so, you need to know the EBS volume ID. You can find it as the `VolumeId` output of the CloudFormation stack, or by running the following:

```
$ aws ec2 describe-volumes --region us-east-1 \
➥ --filters "Name=size,Values=5"  --query "Volumes[].VolumeId" \
➥ --output text
vol-0317799d61736fc5f                ⟵ The output shows the $VolumeId.
```

With the volume ID, you can go on to create a snapshot:

```
$ aws ec2 create-snapshot --region us-east-1 --volume-id $VolumeId      ⟵ Replace with your $VolumeId.
{
  "Description": "",
  "Encrypted": false,
  "VolumeId": "vol-0317799d61736fc5f",
  "State": "pending",                ⟵ Status of your snapshot
  "VolumeSize": 5,
  "StartTime": "2017-09-28T09:00:14.000Z",
  "Progress": "",
  "OwnerId": "486555357186",
  "SnapshotId": "snap-0070dc0a3ac47e21f"   ⟵ Your $SnapshotId
}
```

Creating a snapshot can take some time, depending on how big your volume is and how many blocks have changed since the last backup. You can see the status of the snapshot by running the following:

```
$ aws ec2 describe-snapshots --region us-east-1 --snapshot-ids $SnapshotId
{
  "Snapshots": [
    {
      "Description": "",
      "Encrypted": false,
      "VolumeId": "vol-0317799d61736fc5f",
      "State": "completed",
      "VolumeSize": 5,
      "StartTime": "2017-09-28T09:00:14.000Z",
      "Progress": "100%",
      "OwnerId": "486555357186",
      "SnapshotId": "snap-0070dc0a3ac47e21f"
    }
  ]
}
```

Replace with your $SnapshotId.

The snapshot has reached the state completed.

Progress of your snapshot

Creating a snapshot of an attached, mounted volume is possible, but can cause problems with writes that aren't flushed to disk. You should either detach the volume from your instance or stop the instance first. If you absolutely must create a snapshot while the volume is in use, you can do so safely as follows:

1. Freeze all writes by running `fsfreeze -f /mnt/volume/` on the virtual machine.
2. Create a snapshot and wait until it reaches the pending state.
3. Unfreeze to resume writes by running `fsfreeze -u /mnt/volume/` on the virtual machine.
4. Wait until the snapshot is completed.

Unfreeze the volume as soon as the snapshot reaches the state *pending*. You don't have to wait until the snapshot has finished.

With an EBS snapshot, you don't have to worry about losing data due to a failed EBS volume or human failure. You are able to restore your data from your EBS snapshot.

To restore a snapshot, you must create a new EBS volume based on that snapshot. Execute the following command in your terminal, replacing *$SnapshotId* with the ID of your snapshot.

Adds regularization parameters

```
$ aws ec2 create-volume --region us-east-1 \
➥ --snapshot-id $SnapshotId \
➥ --availability-zone us-east-1a
{
    "AvailabilityZone": "us-east-1a",
    "Encrypted": false,
    "VolumeType": "standard",
    "VolumeId": "vol-0a1afe956678f5f36",
    "State": "creating",
    "SnapshotId": "snap-0dcadf095a785e0bc",
    "CreateTime": "2017-12-07T12:46:13.000Z",
    "Size": 5
}
```

The ID of your snapshot used to create the volume

The $RestoreVolumeId of the volume restored from your snapshot

When you launch an EC2 instance from an AMI, AWS creates a new EBS volume (root volume) based on a snapshot (an AMI includes an EBS snapshot).

Cleaning up AWS OpsWorks

Don't forget to delete the snapshot, the volume, and your stack. The following code will delete the snapshot and volume:

```
$ aws ec2 delete-snapshot --region us-east-1 \
➥ --snapshot-id $SnapshotId
$ aws ec2 delete-volume --region us-east-1 \
➥ --volume-id $RestoreVolumeId
```

Also delete your stack after you finish this section, to clean up all used resources. You created the stack using the UI, so use the UI to delete it. Otherwise, you'll likely be charged for the resources you use.

9.2 Instance store: Temporary block-level storage

An *instance store* provides block-level storage directly attached to the machine hosting your VM. Figure 9.2 shows that the instance store is part of an EC2 instance and available only if your instance is running; it won't persist your data if you stop or terminate the instance. You don't pay separately for an instance store; instance store charges are included in the EC2 instance price.

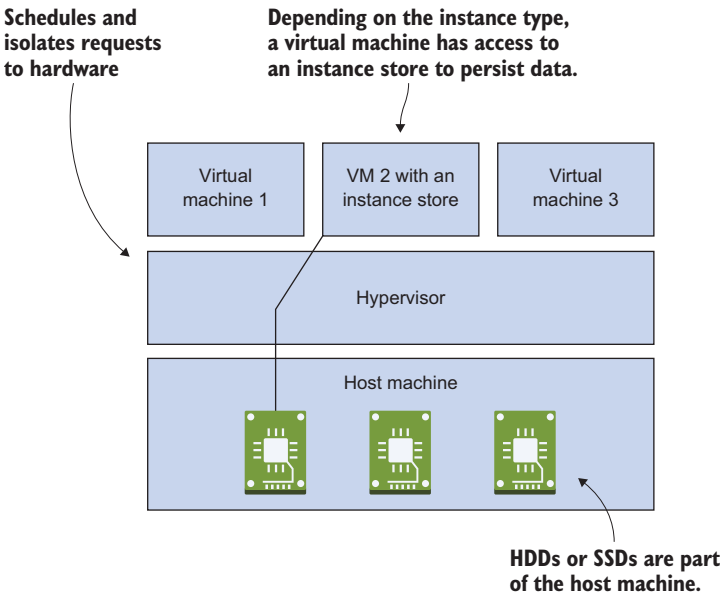


Figure 9.2 The instance store is part of your EC2 instance and uses the host machine's HDDs or SSDs.

In comparison to an EBS volume, which is connected to your VM over the network, the instance store depends upon the VM and can't exist without it. So the instance store will be deleted when you stop or terminate the VM.

Don't use an instance store for data that must not be lost; use it for caching, temporary processing, or applications that replicate data to several servers, as some databases do. If you want to set up your favorite NoSQL database, chances are high that data replication is handled by the application and you can use an instance store.

WARNING If you stop or terminate your EC2 instance, the instance store is lost. *Lost* means all data is destroyed and can't be restored!

AWS offers SSD and HDD instance stores from 4 GB up to 48 TB. Table 9.3 shows a few EC2 instance families providing instance stores.

Table 9.3 Instance families with instance stores

Use case	Instance type	Instance store type	Instance store size in GB
General purpose	m3.medium–m3.2xlarge	SSD	(1 × 4)–(2 × 80)
Compute optimized	c3.large–c3.8xlarge	SSD	(2 × 16)–(2 × 320)
Memory optimized	r3.large–r3.8xlarge	SSD	(1 × 32)–(2 × 320)
Storage optimized	i3.large–i3.16xlarge	SSD	(1 × 950)–(8 × 1,900)
Storage optimized	d2.xlarge–d2.8xlarge	HDD	(3 × 2,000)–(24 × 2,000)

WARNING Starting a virtual machine with instance type m3.medium will incur charges. See https://aws.amazon.com/ec2/pricing/on-demand/ if you want to find out the current hourly price.

To launch an EC2 instance with an instance store manually, open the Management Console and start the Launch Instance wizard as you did in section 3.1. Choose AMI, choose the instance type (m3.medium), and configure the instance details shown in figure 9.3:

1. Click the Add New Volume button.
2. Select instance store 0.
3. Set the device name to dev/sdb.

Tag the instance, configure a security group, and review the stance launch. The instance store can now be used by your EC2 instance.

Listing 9.1 demonstrates how to use an instance store with CloudFormation. If you launch an EC2 instance from an EBS-backed root volume (which is the default), you must define a `BlockDeviceMappings` to map EBS and instance store volumes to device names. Instance stores aren't standalone resources like EBS volumes; the instance store is part of your EC2 instance. Depending on the instance type, you'll have zero, one, or multiple instance store volumes for mapping.

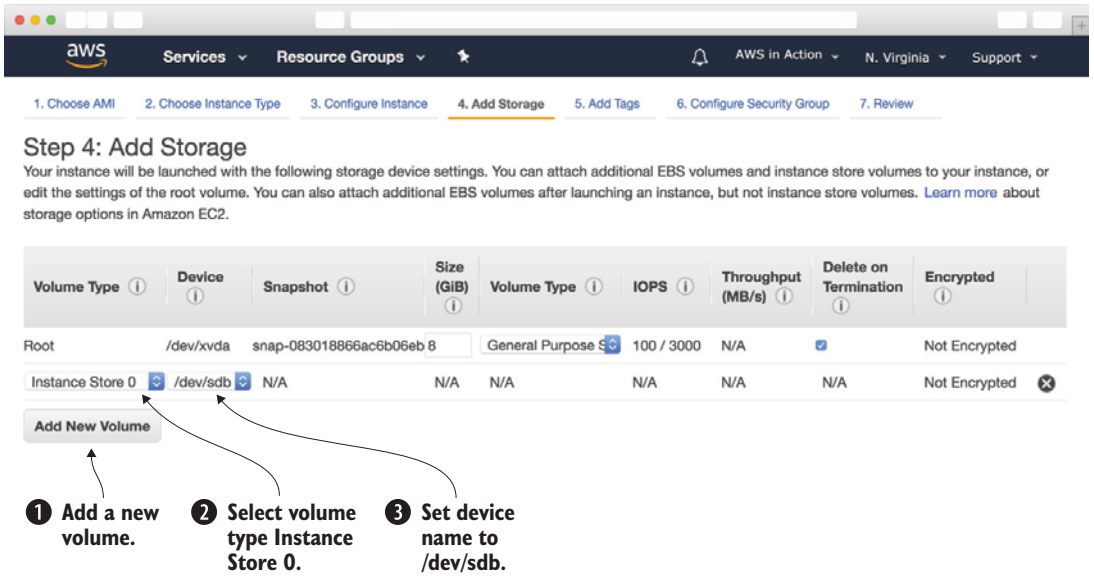


Figure 9.3 Adding an instance store volume manually

Listing 9.1 Connecting an instance store with an EC2 instance with CloudFormation

```
EC2Instance:
  Type: AWS::EC2::Instance
  Properties:
    # [...]
    InstanceType: 'm3.medium'
    BlockDeviceMappings:
    - DeviceName: '/dev/xvda'
      Ebs:
        VolumeSize: '8'
        VolumeType: gp2
    - DeviceName: '/dev/xvdb'
      VirtualName: ephemeral0
```

Choose an instance type with an instance store.

EBS root volume (your OS lives here)

The instance store device will appear as /dev/xvdb.

The instance store is a virtual name like ephemera10 or ephemerall.

Windows-based EC2 instances

The same `BlockDeviceMappings` applies to Windows-based EC2 instances, so the `DeviceName` isn't the same as the drive letter (C:/, D:/, and so on). To go from `DeviceName` to the drive letter, the volume must be mounted. When using Windows, listing 9.1 will still work but the instance store will be available as drive letter Z:/.

Read on to see how mounting works on Linux.

Cleaning up

Don't forget to delete your manually started EC2 instance after you finish this section, to clean up all used resources. Otherwise you'll likely be charged for the resources you use.

9.2.1 Using an instance store

To help you explore instance stores, we created the CloudFormation template located at http://mng.bz/Zu54. Create a CloudFormation stack based on the template by clicking the CloudFormation Quick-Create link at http://mng.bz/V64s.

> **WARNING** Starting a virtual machine with instance type m3.medium will incur charges. See https://aws.amazon.com/ec2/pricing/on-demand/ if you want to find out the current hourly price.

Create a stack based on that template, and select the default VPC and a random subnet. After creating the stack, copy the `PublicName` output to your clipboard and connect via SSH. Usually, instance stores are found at /dev/xvdb to /dev/xvde as shown in the following example:

```
$ sudo fdisk -l
Disk /dev/xvda: 8589 MB [...]
Units = Sektoren of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: gpt


  #         Start          End     Size  Type            Name
  1          4096     16777182       8G  Linux filesyste Linux
128          2048         4095       1M  BIOS boot parti BIOS Boot Partition

Disk /dev/xvdb: 4289 MB [...]
Units = Sektoren of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

EBS device used as root volume containing the OS

The instance store device

To see the mounted volumes, use this command:

```
$ df -h
Filesystem  Size  Used Avail Use% Mounted on
/dev/xvda1  7.8G  1.1G  6.6G  14% /
devtmpfs    1.9G   60K  1.9G   1% /dev
tmpfs       1.9G     0  1.9G   0% /dev/shm
/dev/xvdb   3.9G  1.1G  2.7G  28% /media/ephemeral0
```

The root volume contains the OS.

The instance store volume is mounted automatically.

Your instance store volume is mounted automatically to /media/ephemeral0. If your EC2 instance has more than one instance store volume, ephemeral1, ephemeral2, and so on will be used. Now it's time to run some performance tests to compare the performance of an instance store volume to an EBS volume.

9.2.2 Testing performance

Let's take the same performance measurements as we took in section 9.1.3 to see the difference between the instance store and EBS volumes:

```
$ sudo dd if=/dev/zero of=/media/ephemeral0/tempfile bs=1M count=1024 \
➥ conv=fdatasync,notrunc
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 2.49478 s, 430 MB/s

$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ sudo dd if=/media/ephemeral0/tempfile of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 0.273889 s, 3.9 GB/s
```

6 × write performance compared with EBS from section 9.1.3

60 × read performance compared with EBS from section 9.1.3

Keep in mind that performance can vary, depending on your actual workload. This example assumes a file size of 1 MB. If you're hosting websites, you'll most likely deal with lots of small files instead. The performance characteristics shows that the instance store is running on the same hardware the virtual machine is running on. The volumes are not connected to the virtual machine over the network as with EBS volumes.

Cleaning up

Don't forget to delete your stacks after you finish this section, to clean up all used resources. Otherwise you'll likely be charged for the resources you use.

9.2.3 Backing up your data

There is no built-in backup mechanism for instance store volumes. Based on what you learned in section 8.2, you can use a combination of scheduled jobs and S3 to back up your data periodically:

```
$ aws s3 sync /path/to/data s3://$YourCompany-backup/instancestore-backup
```

But if you need to back up data, you should probably use more durable, block-level storage like EBS. An instance store is better used for ephemeral persistence requirements.

You will learn about another option to store your data in the next chapter: a network file system.

Summary

- Block-level storage can only be used in combination with an EC2 instance because the OS is needed to provide access to the block-level storage (including partitions, file systems, and read/write system calls).
- EBS volumes are connected to a single EC2 instance via network. Depending on your instance type, this network connection can use more or less bandwidth.
- EBS snapshots are a powerful way to back up your EBS volumes to S3 because they use a block-level, incremental approach.
- An instance store is part of a single EC2 instance, and it's fast and cheap. But all your data will be lost if the EC2 instance is stopped or terminated.

10 Sharing data volumes between machines: EFS

This chapter covers

- Creating a highly available shared filesystem
- Mounting your shared filesystem on multiple EC2 instances
- Sharing files between EC2 instances
- Testing performance of your shared filesystem
- Backing up your shared filesystem

Many legacy applications store state in files on disk. Therefore, using Amazon S3, an object store, as described in chapter 8 is not possible by default. Using block storage as discussed in the previous chapter might be an option, but does not allow you to access files from multiple machines in parallel. Hence you need a way to share the files between virtual machines. With Elastic File System (EFS), you can share data between multiple EC2 instances and your data is replicated between multiple *availability zones* (AZ).

EFS is based on the NFSv4.1 protocol, so you can mount it like any other filesystem. In this chapter you learn how to set up EFS, tweak performance, and back your data.

EFS ONLY WORKS WITH LINUX At this time, EFS is not supported by Windows EC2 instances.

Examples are 100% covered by the Free Tier

The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

Let's take a closer look at how EFS works compared to Elastic Block Store (EBS) and the instance store, which were introduced in the previous chapter. An EBS volume is tied to a data center (also called an AZ) and can only be attached over the network to a single EC2 instance from the same data center. Typically EBS volumes are used as the root volumes that contain the operating system, or for relational database systems to store the state. An instance store consists of a hard drive directly attached to the hardware the VM is running on. An instance store can be regarded as ephemeral storage, and is therefore used for caching or for NoSQL databases with embedded data replication only. In contrast, the EFS filesystem can be used by multiple EC2 instances from different data centers in parallel. Additionally, the data on the EFS filesystem is replicated among multiple data centers and remains available even if a whole data center suffers from an outage, which is not true for EBS and instance stores. Figure 10.1 shows the differences.

Now let's take a closer look at EFS. There are two main components to know about:

1. *Filesystem*—Stores your data
2. *Mount target*—Makes your data accessible

The filesystem is the resource that stores your data in an AWS region, but you can't access it directly. To do so, you must create an EFS mount target in a subnet. The mount target provides a network endpoint that you can use to mount the filesystem on an EC2 instance via NFSv4.1. The EC2 instance must be in the same subnet as the EFS mount target, but you can create mount targets in multiple subnets. Figure 10.2 demonstrates how to access the filesystem from EC2 instances running in multiple subnets.

Equipped with the EFS theory about filesystems and mount targets, you can now apply your knowledge to solve a real problem.

Linux is a multiuser operating system. Many users can store data and run programs isolated from each other. Each user can have a home directory that usually is stored

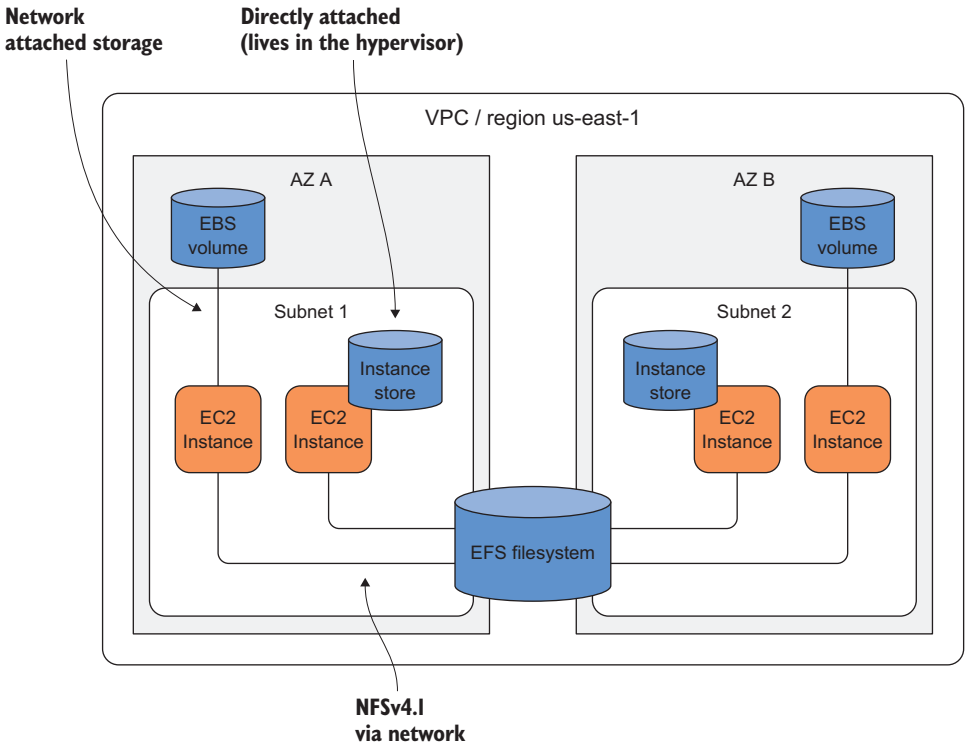


Figure 10.1 Comparing EBS, instance stores, and EFS

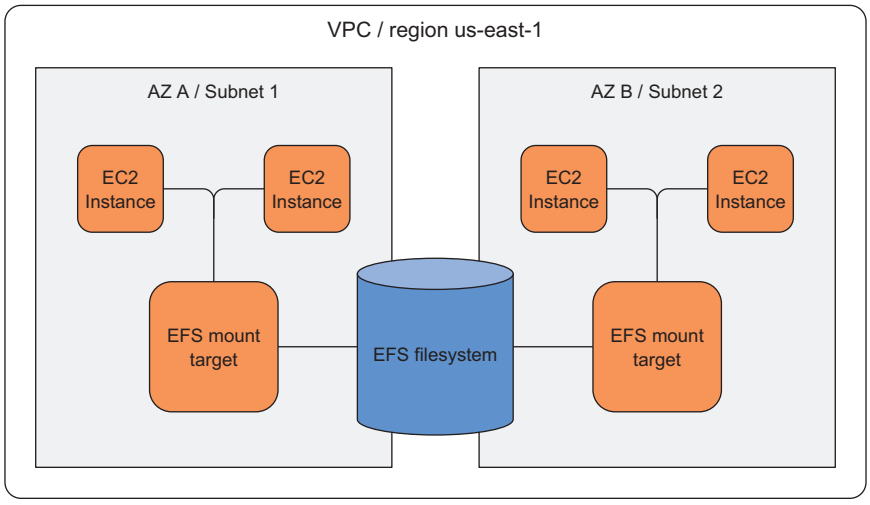
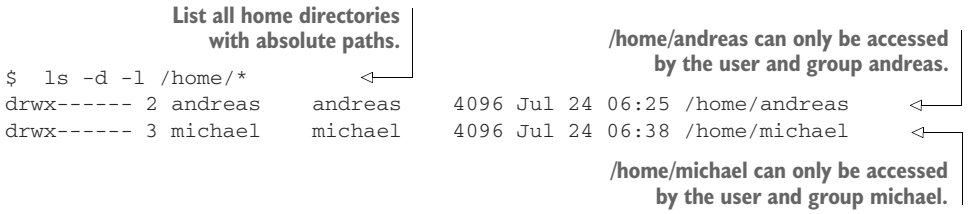


Figure 10.2 Mount targets provide an endpoint for EC2 instances to mount the filesystem in a subnet.

under /home/$username. If the user name is michael, the home directory would be /home/ michael, and only that user would be allowed to read and write in /home/ michael. The `ls -d -l /home/*` command lists all home directories.



If you are using multiple EC2 instances, your users will have a separate home folder on each EC2 instance. If a Linux user uploads a file on one EC2 instance, they can't access the file on another EC2 instance. To solve this problem, create a filesystem and mount EFS on each EC2 instance under /home. The home directories are then shared across all your EC2 instances, and users will feel at home no matter which VM they log in to. In the following sections, you will build this solution step-by-step. First, you will create the filesystem.

10.1 Creating a filesystem

The filesystem is the resource that stores your files, directories, and links. Like S3, EFS grows with your storage needs. You don't have to provision the storage up front. The filesystem is located in an AWS region and replicates your data under the covers across multiple availability zones. You will use CloudFormation to set up the filesystem now.

10.1.1 Using CloudFormation to describe a filesystem

The bare minimum to describe a filesystem resource is shown here.

Listing 10.1 CloudFormation snippet of an EFS filesystem resource

```
Resources:                          <-- Specifies the stack resources
  [...]                                 and their properties
  FileSystem:
    Type: 'AWS::EFS::FileSystem'
    Properties: {}                  <-- Nothing needs to be configured.
```

Optionally, you can add tags to track costs or add other useful meta data with the `FileSystemTags` property.

10.1.2 Pricing

Calculating EFS costs is simple. You only need to know the amount of storage you use in GB. EFS charges you per GB per month. If your EFS filesystem is 5 GB in size, you will be charged 5 GB x 0.30 (USD/GB/month) in us-east-1, for a total of $1.50 USD a month. Please check https://aws.amazon.com/efs/pricing to get the latest pricing

information for your region. The first 5 GB per month are free in the first year of your AWS account (Free Tier).

The filesystem is now described in CloudFormation. To use it, you need to create at least one mount point. Creating a mount target is the subject of the next section.

10.2 Creating a mount target

An EFS mount target makes your data available to EC2 instances via the NFSv4.1 protocol in a single AZ. The EC2 instance communicates with the mount target via a TCP/IP network connection. As you learned in section 6.4, security groups are how you control network traffic on AWS. You can use a security group to allow inbound traffic to an EC2 instance or an RDS database, and the same is true for a mount target. Security groups control which traffic is allowed to enter the mount target. The NFS protocol uses port 2049 for inbound communication. Figure 10.3 shows how mount targets are protected.

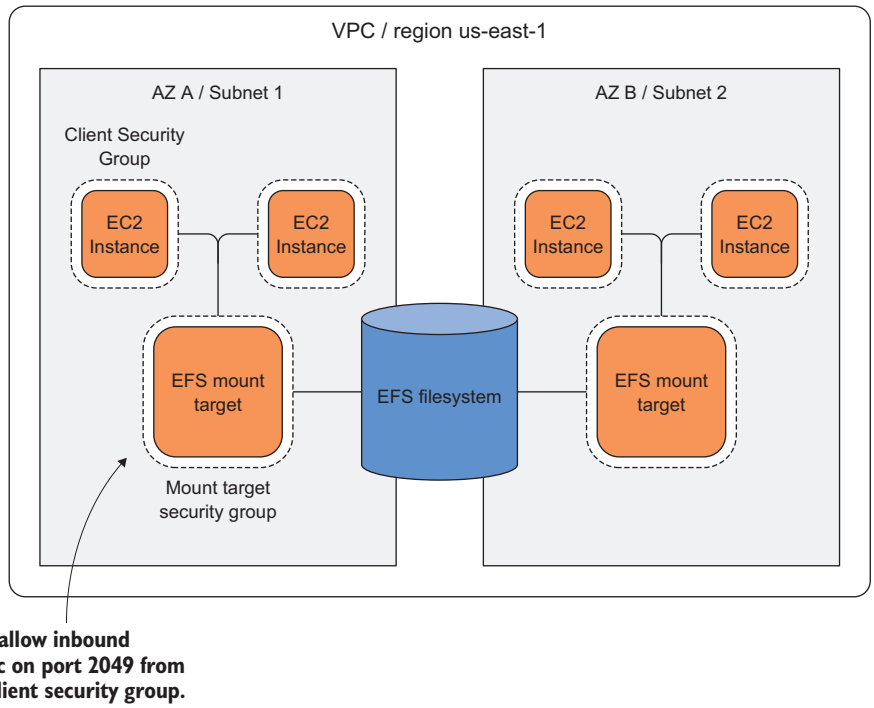


Figure 10.3 EFS mount targets are protected by security groups.

In our example, to control traffic as tightly as possible, you won't white list IP addresses. Instead, you'll create two security groups. The client security group will be attached to all EC2 instances that want to mount the filesystem. The mount target security group allows inbound traffic on port 2049 only for traffic that comes from the

client security group. This way, you can have a dynamic fleet of clients who are allowed to send traffic to the mount targets. You used the same approach for the SSH bastion host in section 6.4.

You can use CloudFormation to manage an EFS mount target. The mount target references the filesystem, needs to be linked to a subnet, and is also protected by at least one security group. You will first describe the security groups, followed by the mount target, as shown in listing 10.2.

Listing 10.2 CloudFormation snippet of an EFS mount target and security groups

```
Resources:
  [...]
  EFSClientSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'EFS Mount target client'
      VpcId: !Ref VPC
  MountTargetSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'EFS Mount target'
      SecurityGroupIngress:
      - FromPort: 2049
        IpProtocol: tcp
        SourceSecurityGroupId: !Ref EFSClientSecurityGroup
        ToPort: 2049
      VpcId: !Ref VPC
  MountTargetA:
    Type: 'AWS::EFS::MountTarget'
    Properties:
      FileSystemId: !Ref FileSystem
      SecurityGroups:
      - !Ref MountTargetSecurityGroup
      SubnetId: !Ref SubnetA
```

The client security group needs no rules. It's just used to mark traffic.

Allow traffic on port 2049.

Only allow traffic from the client security group.

Connect mount target with the filesystem.

Assign the security group.

Connect with a subnet which also determines the AZ.

Copy the MountTargetA resource and also create a mount target for SubnetB.

```
Resources:
  [...]
  MountTargetB:
    Type: 'AWS::EFS::MountTarget'
    Properties:
      FileSystemId: !Ref FileSystem
      SecurityGroups:
      - !Ref MountTargetSecurityGroup
      SubnetId: !Ref SubnetB
```

The other subnet is used.

The mount targets can now be used in the next section, where you finally mount the /home directory.

10.3 Mounting the EFS share on EC2 instances

EFS creates a DNS name for each filesystem following the schema *$FileSystemID*.efs.*$Region*.amazonaws.com. Inside an EC2 instance, this name resolves to the mount target of the instance's AZ. AWS suggests the following mount options:

- `nfsvers=4.1`—Specifies which version of the NFS protocol to use.
- `rsize=1048576`—Read data block size, in bytes, to be transferred at one time.
- `wsize=1048576`—Write data block size, in bytes, to be transferred at one time.
- `hard`—If the EFS share is down, wait for the share to come back online.
- `timeo=600`—The time in deciseconds (tenths of a second) the NFS client waits for a response before it retries an NFS request.
- `retrans=2`—The number of times the NFS client retries a request before it attempts further recovery action.

This snippet shows the full mount command:

```
$ mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,\
➥ retrans=2 $FileSystemID.efs.$Region.amazonaws.com:/ $EFSMountPoint
```

Replace *`$FileSystemID`* with the EFS filesystem, such as fs-123456. Replace *`$Region`* with the region, such as us-east-1, and *`$EFSMountPoint`* with the local path where the filesystem is mounted. You can also use the /ets/fstab file to automatically mount on startup:

```
$FileSystemID.efs.$Region.amazonaws.com:/ $EFSMountPoint nfs4 nfsvers=4.1,
➥ rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,_netdev 0 0
```

To make sure that the DNS name can be resolved and the other side is listening to the port, you can use the following Bash script to wait until the mount target is ready:

```
$ while ! nc -z $FileSystemID.efs.$Region.amazonaws.com 2049;
➥ do sleep 10; done
$ sleep 10
```

To use the filesystem, you have to mount it on EC2 instances using one of the mount targets that you created. It's now time to add two EC2 instances to the CloudFormation template. Each EC2 instance should be placed in a different subnet and mount the filesystem to /home. The /home directory will exist on both EC2 instances, and it will also contain some data (such as the folder ec2-user). You have to ensure that you're copying the original data the first time before you mount the EFS filesystem, which is empty by default. This listing describes the EC2 instance that copies the existing /home folder before the shared home folder is mounted.

Listing 10.3 An EC2 instance in SubnetA and security group resources

```
Resources:
  [...]
  EC2SecurityGroup:                                  <-- Security Group to allow SSH traffic from the internet
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'EC2 instance'
      SecurityGroupIngress:
      - CidrIp: '0.0.0.0/0'
        FromPort: 22
        IpProtocol: tcp
        ToPort: 22
      VpcId: !Ref VPC
  EC2InstanceA:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: 't2.micro'
      KeyName: mykey
      NetworkInterfaces:
      - AssociatePublicIpAddress: true               <-- Ensure the EC2 instance gets a public IP address for SSH access.
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref EC2SecurityGroup                      <-- Attach the security group to allow SSH.
        - !Ref EFSClientSecurityGroup                <-- Attach the client mount target security group.
        SubnetId: !Ref SubnetA                       <-- Place instance into subnet A.
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          bash -ex << "TRY"
            while ! nc -z ${FileSystem}.efs.${AWS::Region}.amazonaws.com 2049;
          ➥ do sleep 10; done                       <-- Wait until filesystem is available.
            sleep 10
            mkdir /oldhome                           <-- Create temporary folder for /home content.
            cp -a /home/. /oldhome                   <-- Copy existing /home to /oldhome and preserve permissions (-a).
            echo "${FileSystem}.efs.${AWS::Region}.amazonaws.com:/ /home nfs4
          ➥ nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,
          ➥retrans=2,_netdev 0 0" >> /etc/fstab
            mount -a                                 <-- Mount filesystem.
            cp -a /oldhome/. /home                   <-- Copy /oldhome to new /home.
          TRY
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
          ➥ --resource EC2InstanceA --region ${AWS::Region}   <-- Let CloudFormation know that the EC2 instance resource is efs-with-backup.
      CreationPolicy:
        ResourceSignal:
          Timeout: PT10M
      DependsOn:                                     <-- Wait for the internet gateway.
      - VPCGatewayAttachment
      - MountTargetA                                 <-- Wait for the mount target.
```

After the EC2 instance is launched, you will find the first data on the EFS share. The second EC2 instance is similar but in a different subnet, and does not copy the existing

/home content because this was already done by the previous EC2 instance. Here are the details.

Listing 10.4 An EC2 instance in SubnetB and security group resources

```
Resources:
  [...]
  EC2InstanceB:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: 't2.micro'
      KeyName: mykey
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref EC2SecurityGroup
        - !Ref EFSClientSecurityGroup
        SubnetId: !Ref SubnetB          <-- Place into the other subnet.
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          bash -ex << "TRY"
            while ! nc -z ${FileSystem}.efs.${AWS::Region}.amazonaws.com 2049;
➥ do sleep 10; done
            sleep 10
            echo "${FileSystem}.efs.${AWS::Region}.amazonaws.com:/ /home nfs4
➥ nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,
➥ retrans=2,_netdev 0 0" >> /etc/fstab
            mount -a                                          <-- The old /home is not copied here.
          TRY                                                     This is already done on the first EC2
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}   instance in subnet A.
➥ --resource EC2InstanceB --region ${AWS::Region}
    CreationPolicy:
      ResourceSignal:
        Timeout: PT10M
    DependsOn:
    - VPCGatewayAttachment
    - MountTargetB
```

To make things easier, you can also add outputs to the template to expose the public IP addresses of your EC2 instances like this:

```
Outputs:
  EC2InstanceAIPAddress:
    Value: !GetAtt 'EC2InstanceA.PublicIp'
    Description: 'EC2 Instance (AZ A) public IP address (connect via SSH)'
  EC2InstanceBIPAddress:
    Value: !GetAtt 'EC2InstanceB.PublicIp'
    Description: 'EC2 Instance (AZ B) public IP address (connect via SSH)'
```

The CloudFormation template is now complete. It contains:

- Filesystem (EFS)
- Two EFS mount targets in SubnetA and SubnetB
- Security groups to control traffic to the mount targets
- EC2 instances in both subnets, including a `UserData` script to mount the filesystem

It's now time to create a stack based on your template, to create all the resources in your AWS account. You can find the full code for the template at /chapter10/template.yaml in the book's code folder. You can use the AWS CLI to create the stack:

```
$ aws cloudformation create-stack --stack-name efs \
➥ --template-url https://s3.amazonaws.com/awsinaction-code2/\
➥ chapter10/template.yaml
```

> **Where is the template located?**
> You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at chapter10/template.yaml. On S3, the same file is located at http://mng.bz/XIUE.

Once the stack is in the state CREATE_COMPLETE, you have two EC2 instances running. Both mounted the EFS share to /home. You also copied the old /home data to the EFS share. It's time to connect to the instances via SSH and do some tests (in the next section), to see if users can really share files between the EC2 instances in their home directory.

10.4 Sharing files between EC2 instances

Open an SSH connection to the virtual machine in subnet A. You can use the AWS CLI to get the stack output, from which you can get the public IP address:

```
$ aws cloudformation describe-stacks --stack-name efs \
➥ --query "Stacks[0].Outputs"
[{
  "Description": "[...]",
  "OutputKey": "EC2InstanceAIPAddress",
  "OutputValue": "54.158.102.196"
}, {
  "Description": "[...]",
  "OutputKey": "EC2InstanceBIPAddress",
  "OutputValue": "34.205.4.174"
}]
```

Use the SSH key `mykey` to authenticate, and replace *`$PublicIpAddress`* with the IP address of the `EC2InstanceAIPAddress` output from the stack:

```
$ ssh -i $PathToKey/mykey.pem ec2-user@$PublicIpAddress
```

Open a second SSH connection to the virtual machine in subnet B. Use the same command you just used, but this time, replace `$PublicIpAddress` with the IP address of the `EC2InstanceBIPAddress` output from the stack.

You now have two SSH connections open. In both SSH sessions you should be located in the /home/ec2-user folder. Check if this is true on both machines:

```
$ pwd
/home/ec2-user
```

The output confirms that your current directory is /home/ec2-user.

Also check if there are any files or folders in /home/ec2-user:

```
$ ls
```

If no data is returned, the folder /home/ec2-user is empty.

Now, create a file on one of the machines:

```
$ touch i-was-here
```

touch creates an empty file.

On the other machine, confirm that you can see the new file:

```
$ ls
i-was-here
```

You can see the file.

You now have access to the same home directory on both machines. You could add hundreds of machines to this example. All would share the same home directory, and your users would be able to access the same home directory on all EC2 instances. You can apply the same mechanism to share files between a fleet of web servers (for example, the `/var/www/html` folder), or to design a highly available Jenkins server (such as `/var/lib/jenkins`).

To operate the solution successfully, you also need to take care of backups, performance tuning, and monitoring. You will learn about this in the following sections.

10.5 Tweaking performance

To compare EFS with other storage options, we'll use the same simple performance test that we used in section 9.1.3 to test the performance of an EBS volume. The tool `dd` can perform block-level reads and writes.

```
$ sudo dd if=/dev/zero of=/home/ec2-user/tempfile bs=1M count=1024 \
➥ conv=fdatasync,notrunc
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 10.4138 s, 103 MB/s

$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3

$ sudo dd if=/home/ec2-user/tempfile of=/dev/null bs=1M count=1024
```

Writes 1 MB 1,024 times

103 MB/s write performance

Flushes caches

Reads 1 MB 1,024 times

```
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 10.2916 s, 104 MB/s
```

104MB/s read performance

Please keep in mind that this performance tests assumes files of 1 MB. Depending on your workload, files can be smaller or bigger, which leads to different results. Usually, throughput will degrade for smaller files. Comparing these numbers to the `dd` results on EBS and instance store is only of limited informative value.

10.5.1 Performance mode

So far, you've used the General Purpose performance mode, which is fine for most workloads, especially latency-sensitive ones where small files are served most of the time. The `/home` directory is a perfect example of such a workload. Users are not constantly opening and saving files. Instead they list files from time to time and then open a specific file. When the user opens a file, they expect low latency to get the file instantaneously.

But sometimes, EFS is used to store massive amounts of data for analytics. For data analytics, latency is not important. Throughput is the metric you want to optimize instead. If you want to analyze gigabytes or terabytes of data, it doesn't matter if your time to first byte takes 1 ms or 100 ms. Even a small increase in throughput will decrease the time it will take to analyze the data. For example, analyzing 1 TB of data with 100 MB/sec throughput will take 174 minutes. That's almost three hours, so the first few milliseconds don't really matter. Optimizing for throughput can be achieved using the Max I/O performance mode. The performance mode being used by an EFS filesystem cannot be changed—you set it when the filesystem is created. Therefore, to change the performance mode, you have to create a new filesystem. We recommend you start with the General Purpose performance mode if you are unsure which mode fits best for your workload. You will learn how to check whether you made the right decision by looking at monitoring data in the following section.

10.5.2 Expected throughput

The performance of EFS scales with the amount of storage you use. EFS also allows for bursting, because many workloads require high performance for a short period of time and are idle for most of the rest of the time.

The baseline rate is 51.2 KB/s per 1.1 GB of storage (or 52.4 MB/s per 1100 GB). You can burst 50% of the time if the filesystem is not accessed for the remaining time. Table 10.1 shows how you can calculate your burst rate.

Table 10.1 EFS burst rate

filesystem size	Burst rate
< 1100 GB	104.9 MB/s
>= 1100 GB	104.9 MB/s per 1100 GB of data stored

The burst rate is exactly the rate we measured in the simple performance test at the beginning of this section! To get the exact numbers for how long you can burst, you can consult the official documentation: "Throughput Scaling in Amazon EFS" at http://mng.bz/Y5Q9.

The throughput rules are complicated. Luckily, you can use CloudWatch to observe the numbers and send an alert if you run out of credits. That's the topic of the next section.

10.6 Monitoring a filesystem

CloudWatch is the AWS service that stores all kinds of metrics. EFS sends useful metrics, the most important of which to watch are:

- `BurstCreditBalance`—The current credit balance. You need credits to be able to burst throughput to your EFS filesystem.
- `PermittedThroughput`—The throughput you have at this moment. Takes burst credits and size into account.
- `Read/Write/Metadata/TotalIOBytes`—Information in bytes about reads, writes, metadata, and total I/O usage.
- `PercentIOLimit`—How close you are to hitting the I/O limit. If this metric is at 100%, the Max I/O performance mode would be a good choice.

Let's look at those metrics in more detail now. You will also get some hints about useful thresholds to define alarms on those metrics.

10.6.1 Should you use Max I/O Performance mode?

The `PercentIOLimit` metric shows how close a filesystem is to reaching the I/O limit of the General Purpose performance mode (not used for other modes). If this metric is at 100% for more than half of the time, you should consider creating a new filesystem with Max I/O performance mode enabled. It makes sense to create an alarm on this metric, as shown in figure 10.4. To create a CloudWatch Alarm in the Management Console:

1. Open the CloudWatch Management Console: https://us-east-1.console.aws.amazon.com/cloudwatch/home.
2. Click the Alarms link on the left.
3. Click the Create Alarm button.
4. Under EFS Metrics, click File System Metrics.
5. Select the PercentIOLimit metric.
6. Click the Next button.
7. Fill out the fields as shown in figure 10.4.

You might set an alarm to trigger if the 15-minute average of the metric is higher than 95% for 4 out of 4 datapoints. The alarm action usually sends a message to an SNS topic which you can subscribe to via email.

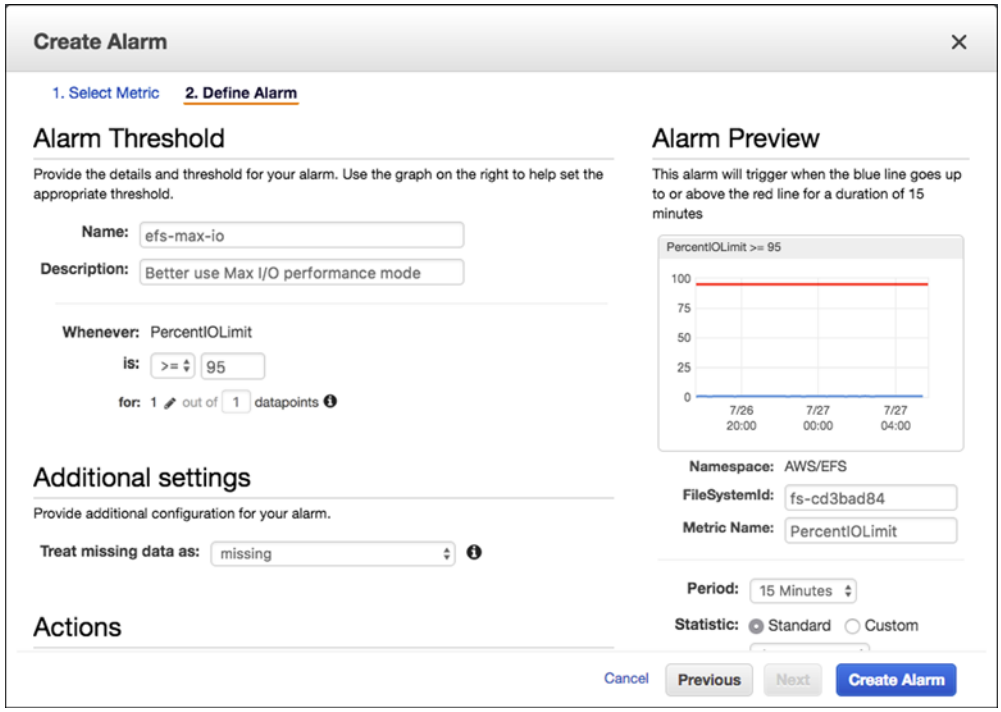


Figure 10.4 Creating a CloudWatch Alarm on EFS's PercentIOLimit metric

10.6.2 Monitoring your permitted throughput

In the previous section, we promised that there is an easier way to get access to the actual throughput of your filesystem besides doing the math. The `PermittedThroughput` metric provides this important information. It takes into account the size of your filesystem and your credit balance to calculate the permitted throughput of your filesystem. Since your credit balance changes all the time (you either consume credits or new credits are added), the permitted throughput can be volatile. Figure 10.5 shows a line chart of the `PermittedThroughput` and the `BurstCreditBalance` at the moment when you run out of credits.

If you rely on credits to get the expected performance, you should create an alarm that monitors your `BurstCreditBalance`. You might set an alarm to trigger if the 10-minute average of the `BurstCreditBalance` metric is lower than 192 GB for one consecutive period, which is the last hour where you can burst at 100 MB/sec. Don't set the threshold too low: you need some time to react! (You can add dummy files to increase the EFS filesystem size, which increases throughput.)

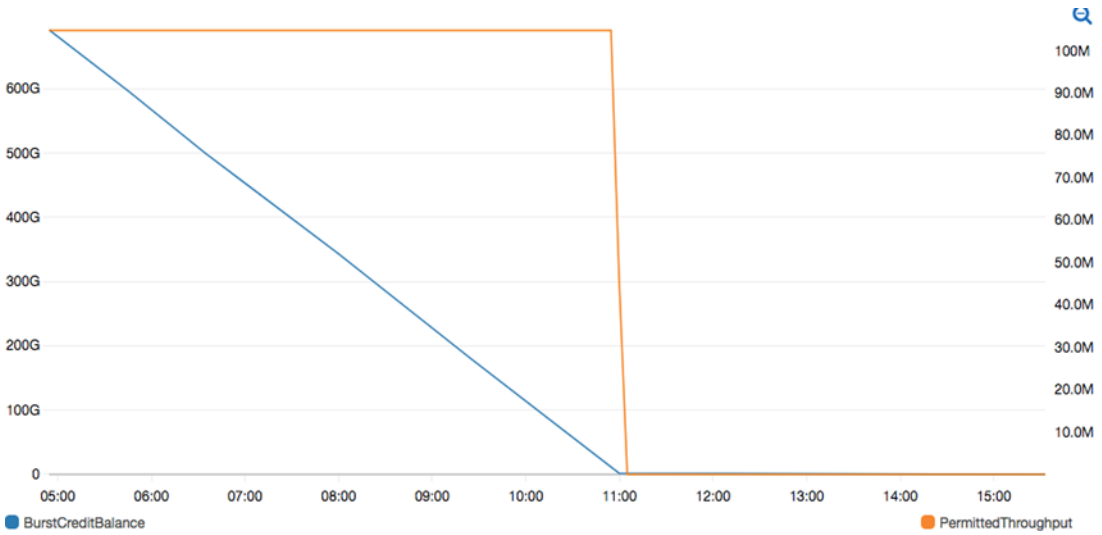


Figure 10.5 Creating a CloudWatch Alarm on EFS's `PermittedThroughout` metric

Listing 10.5 shows a CloudFormation snippet for an alarm to monitor the `BurstCreditBalance`.

Listing 10.5 Alarm resource on the `BurstCreditBalance` metric

```
Resources:
  [...]
  FileSystemBurstCreditBalanceTooLowAlarm:
    Type: 'AWS::CloudWatch::Alarm'
    Properties:
      AlarmDescription: 'EFS file system is running out of burst credits.'
      Namespace: 'AWS/EFS'
      MetricName: BurstCreditBalance
      Statistic: Average
      Period: 600
      EvaluationPeriods: 1
      ComparisonOperator: LessThanThreshold
      Threshold: 192416666667
      AlarmActions:
      - 'arn:aws:sns:us-east-1:123456789012:SNSTopicName'
      Dimensions:
      - Name: FileSystemId
        Value: !Ref FileSystem
```

Name of the metric

192 GB in bytes (last hour when you can burst at 100 MB/sec)

SNS topic ARN to send the alert to. You can also define a SNS topic resource in your template and reference it here.

10.6.3 Monitoring your usage

Access to EFS are either reads, writes, or metadata (metadata is not included in reads or writes). Metadata could be the size and ownership information about a file, or it could be locks to avoid concurrent access to a file. A stacked area chart in CloudWatch can give you a good overview of all the activity. Figure 10.6 shows such a chart.

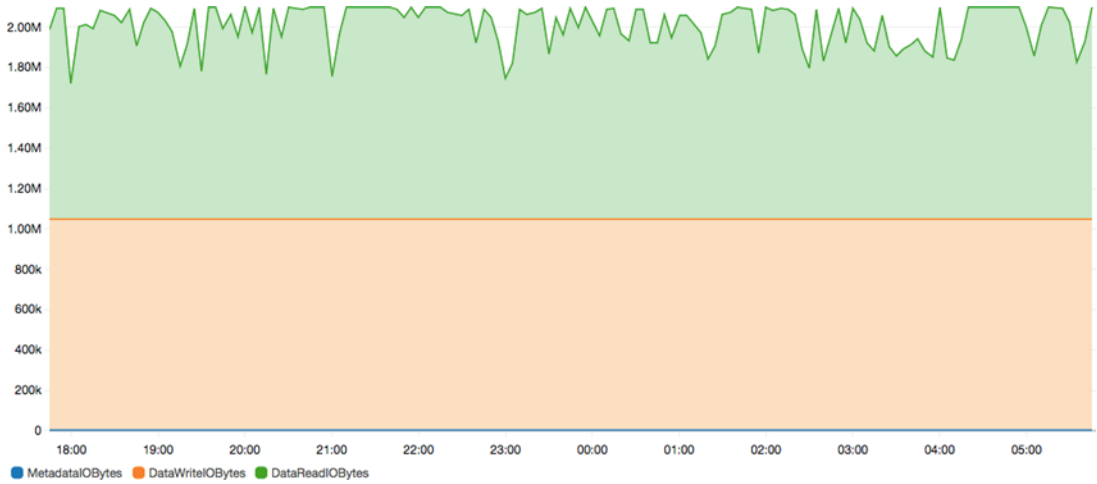


Figure 10.6 CloudWatch graph on EFS usage

Creating an alarm on the usage data makes sense if you know your workload very well. Otherwise you should be safe enough with the alarms from the previous metrics.

10.7 Backing up your data

EFS stores all your files on multiple disks in multiple availability zones. Thus the chances of losing data because of a hardware issue are low. But what about a human error like `rm -rf /`, which removes all the files on Linux (including files in mounted folders)? Or an application bug that corrupts data? Unfortunately, EFS does not provide a native way to back up your EFS filesystem at the time of writing. But there are multiple options to create your own backup solution:

- Sync the files to S3 from time to time.
- Sync the files to an EBS volume from time to time, and create a snapshot of the volume after each sync.
- Sync the files to another EFS filesystem.
- Use a third-party backup solution. EFS is just another volume on your operating system.

If you need a cost effective way to back up the history of changes to EFS, we recommend you use an EBS volume and snapshots. As you learned in section 9.1, EBS snapshots are block-level incremental backups. This means only changes to the blocks are stored when performing multiple snapshots. One disadvantage of this solution is that your data must be small enough to fit on a single EBS volume. If it's not, consider using a second EFS filesystem for your backups.

To implement the EBS backup strategy, you need to add an EBS volume, and attach it to one of the EC2 instances. Finally, you have to implement the logic to synchronize

the mounted filesystem with EBS and trigger a snapshot. Let's start by adding the EBS volume.

10.7.1 Using CloudFormation to describe an EBS volume

The EBS volume needs to be defined in the CloudFormation template. You also need to configure that the volume should be attached to the EC2 instance in subnet A. Listing 10.6 shows the CloudFormation snippet.

Listing 10.6 EBS volume resource attached to an EC2 instance

```
Resources:
  [...]
  EBSBackupVolumeA:
    Type: 'AWS::EC2::Volume'
    Properties:
      AvailabilityZone: !Select [0, !GetAZs '']
      Size: 5
      VolumeType: gp2
  EBSBackupVolumeAttachmentA:
    Type: 'AWS::EC2::VolumeAttachment'
    Properties:
      Device: '/dev/xvdf'
      InstanceId: !Ref EC2InstanceA
      VolumeId: !Ref EBSBackupVolumeA
```

5 GB in size (you can increase this)

The EBS volume needs to be in the same AZ as the EC2 instance.

Use the SSD backed general purpose storage type.

The EC2 instance as one side of the volume attachment

The EBS volume as the other side of the volume attachment.

Now, the volume is connected to the EC2 instance in the CloudFormation template.

10.7.2 Using the EBS volume

It might take a moment before the operating system can see the EBS volume as a new disk. A plain new EBS volume is not yet formatted, so you have to format the disk with a filesystem at first use. After that, you can mount the disk. You will use a cron job that runs every 15 minutes to:

1. Copy the files from /home (EFS) to /mnt/backup (EBS) using rsync.
2. Freeze the backup mount, to prevent any additional writes using the `fsfreeze` command.
3. Start the snapshot creation using the AWS CLI.
4. Unfreeze the backup mount using the `fsfreeze -u` command.

The following listing extends the user data script executed by the EC2 instance on startup.

Listing 10.7 Mount the EBS volume and back up the data from EFS periodically

```
[...]
/opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName}
➥ --resource EC2InstanceA --region ${AWS::Region}

while ! [ "`fdisk -l | grep '/dev/xvdf' | wc -l`" -ge "1" ]; do
```

Wait until EBS volume is attached.

```
  sleep 10
done

if [[ "`file -s /dev/xvdf`" != *"ext4"* ]]; then
  mkfs -t ext4 /dev/xvdf
fi

mkdir /mnt/backup
echo "/dev/xvdf /mnt/backup ext4 defaults,nofail 0 2" >> /etc/fstab
mount -a

cat > /etc/cron.d/backup << EOF
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/opt/aws/bin
MAILTO=root
HOME=/
*/15 * * * *
➥ root rsync -av --delete /home/ /mnt/backup/ ;
➥ fsfreeze -f /mnt/backup/ ;
➥ aws --region ${AWS::Region} ec2 create-snapshot
➥ --volume-id ${EBSBackupVolumeA} --description "EFS backup" ;
➥ fsfreeze -u /mnt/backup/
EOF
```

Format EBS volume if needed.

Mount EBS volume.

Install backup cron job.

To allow the EC2 instance to create a snapshot of its own EBS volume, you have to add an instance profile with a IAM role that allows the `ec2:CreateSnapshot` action. Listing 10.8 shows the CloudFormation snippet.

Listing 10.8 CloudFormation snippet of an EC2 Instance Profile resource

```
Resources:
  [...]
  InstanceProfile:
    Type: 'AWS::IAM::InstanceProfile'
    Properties:
      Roles:
      - !Ref Role
  Role:
    Type: 'AWS::IAM::Role'
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Principal:
            Service: 'ec2.amazonaws.com'
          Action: 'sts:AssumeRole'
      Policies:
      - PolicyName: ec2
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
          - Effect: Allow
            Action: 'ec2:CreateSnapshot'
            Resource: '*'
```

Last but not least, you have to attach the instance profile to the EC2 instance by modifying the existing `EC2InstanceA` resource.

```
Resources:
  [...]
  EC2InstanceA:
    Type: 'AWS::EC2::Instance'
    Properties:
      IamInstanceProfile: !Ref InstanceProfile
      [...]
```

It's now time to test the new stack. Don't forget to delete the old stack first. Then, create a new stack based on the extended template. You can find the full code for the template at /chapter10/template-with-backup.yaml in the book's code folder.

```
$ aws cloudformation delete-stack --stack-name efs
$ aws cloudformation create-stack --stack-name efs-with-backup \
➥ --template-url https://s3.amazonaws.com/awsinaction-code2/\
➥ chapter10/template-with-backup.yaml \
➥ --capabilities CAPABILITY_IAM
```

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at chapter10/template-with-backup.yaml. On S3, the same file is located at http://mng.bz/K87P.

When the stack is in the CREATE_COMPLETE state, you have two EC2 instances running. Every 15 minutes, a new EBS snapshot will be created. You need to be patient to validate that the snapshots are working.

Cleaning up

It's time to delete the running CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name efs-with-backup
```

Also use the Management Console to delete all EBS snapshots that have been created as periodical backups.

You now have your own backup solution implemented. Keep in mind that the EBS volume does not automatically grow with your EFS filesystem. You have to adjust the size of the EBS volume manually.

Summary

- EFS provides a NFSv4.1-compliant filesystem that can be shared between Linux EC2 instances in different availability zones.
- EFS mount targets are bound to an availability zone and are protected by security groups.
- You need at least two mount targets in different AZs for high availability.
- EFS does not provide snapshots for point-in-time recovery.
- Data that is stored in EFS is replicated across multiple AZs.

11 Using a relational database service: RDS

This chapter covers

- Launching and initializing relational databases with RDS
- Creating and restoring database snapshots
- Setting up a highly available database
- Tweaking database performance
- Monitoring databases

Relational databases are the de facto standard for storing and querying structured data, and many applications are built on top of a relational database system such as MySQL. Typically, relational databases focus on data consistency and guarantee ACID database transactions (atomicity, consistency, isolation, and durability). A typical task is storing and querying structured data like the accounts and transactions in an accounting application.

If you want to use a relational database on AWS, you have two options:

- Use the managed relational database service Amazon RDS, which is offered by AWS.
- Operate a relational database yourself on top of virtual machines.

The Amazon Relational Database Service (Amazon RDS) offers ready-to-use relational databases such as PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server. If your application supports one of these relational database systems, the migration to Amazon RDS is easy.

Beyond that, AWS offers its own engine called Amazon Aurora, which is MySQL- and PostgreSQL-compatible. If your application supports MySQL or PostgreSQL, the migration to Amazon Aurora is easy.

RDS is a managed service. The managed service provider—in this case AWS—is responsible for providing a defined set of services—in this case, operating a relational database system. Table 11.1 compares using an RDS database and hosting a database yourself on virtual machines.

Table 11.1 Managed service RDS vs. a self-hosted database on virtual machines

	Amazon RDS	Self-hosted on virtual machines
Cost for AWS services	Higher because RDS costs more than virtual machines (EC2)	Lower because virtual machines (EC2) are cheaper than RDS
Total cost of ownership	Lower because operating costs are split among many customers	Much higher because you need your own manpower to manage your database
Quality	AWS professionals are responsible for the managed service.	You'll need to build a team of professionals and implement quality control yourself.
Flexibility	High, because you can choose a relational database system and most of the configuration parameters	Higher, because you can control every part of the relational database system you installed on virtual machines

You'd need considerable time and know-how to build a comparable relational database environment based on virtual machines, so we recommend using Amazon RDS for relational databases whenever possible to decrease operational costs and improve quality. That's why we won't cover hosting your own relational database on VMs in this book. Instead, we'll introduce Amazon RDS in detail.

In this chapter, you'll launch a MySQL database with the help of Amazon RDS. Chapter 2 introduced a WordPress setup like the one shown in figure 11.1; you'll reuse this example in this chapter, focusing on the database part. After the MySQL database is up and running, you'll learn how to import, back up, and restore data. More advanced topics like setting up a highly available database and improving the performance of the database will follow.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

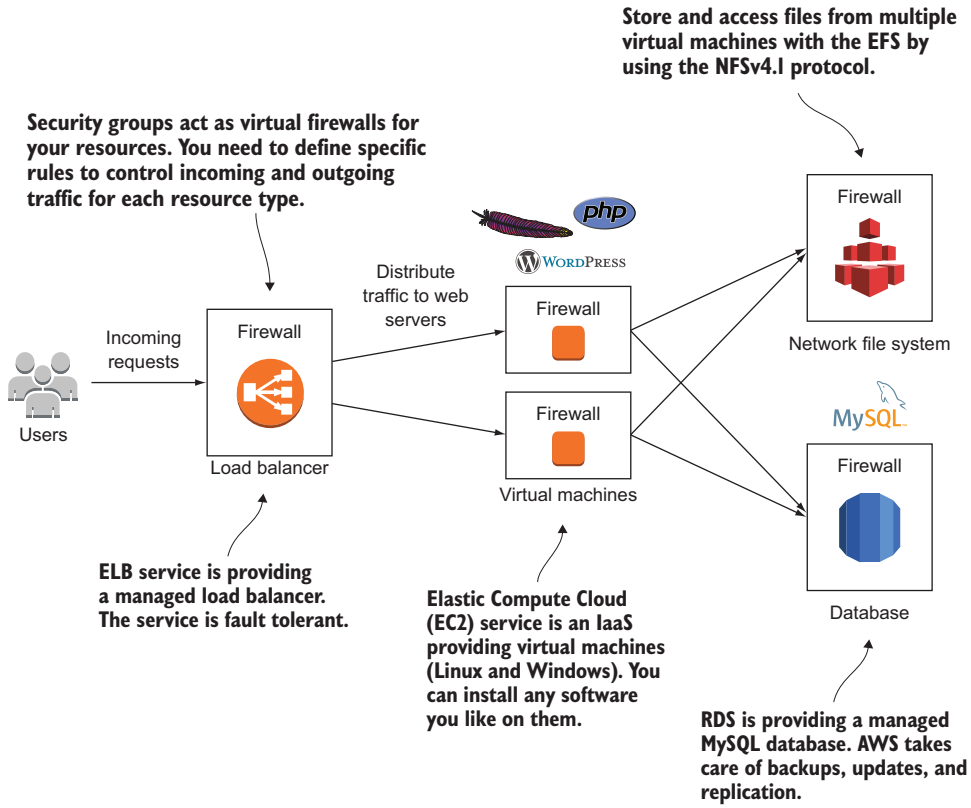


Figure 11.1 The company's blogging infrastructure consists of two load-balanced web servers running WordPress and a MySQL database server.

All the examples in this chapter use a MySQL database used by a WordPress application. You can easily transfer what you learn to other database engines such as Aurora, PostgreSQL, MariaDB, Oracle Database, Microsoft SQL Server, and to applications other than WordPress.

11.1 Starting a MySQL database

The popular blogging platform WordPress is built on top of a MySQL relational database. If you want to host a WordPress blog on your VM, you'll need to run the PHP application—for example, with the help of an Apache web server—and you'll need to operate a MySQL database where WordPress stores the articles, comments, and author accounts. Amazon RDS offers a MySQL database as a managed service, so you no longer need to install, configure, and operate a MySQL database yourself.

11.1.1 Launching a WordPress platform with an RDS database

Launching a database consists of two steps:

1. Launching a database instance
2. Connecting an application to the database endpoint

To set up a WordPress blogging platform with a MySQL database, you'll use the same CloudFormation template you used in chapter 2. You also used Amazon RDS there. The template can be found on GitHub and on S3. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at chapter11/template.yaml. On S3, the same file is located at http://mng.bz/a4C8.

Execute the following command to create a CloudFormation stack containing an RDS database instance with a MySQL engine and web servers serving the WordPress application:

```
$ aws cloudformation create-stack --stack-name wordpress --template-url \
➥ https://s3.amazonaws.com/awsinaction-code2/chapter11/template.yaml \
➥ --parameters ParameterKey=KeyName,ParameterValue=mykey \
➥ ParameterKey=AdminPassword,ParameterValue=test1234 \
➥ ParameterKey=AdminEMail,ParameterValue=your@mail.com
```

You'll wait several minutes while the CloudFormation stack is created in the background, which means you'll have enough time to learn the details of the RDS database instance while the template is launching. Listing 11.1 shows parts of the CloudFormation template used to create the `wordpress` stack. Table 11.2 shows the attributes you need when starting an RDS database using CloudFormation or the Management Console.

Table 11.2 Attributes needed to connect to an RDS database

Attribute	Description
`AllocatedStorage`	Storage size of your database in GB
`DBInstanceClass`	Size (also known as instance type) of the underlying virtual machine
`Engine`	Database engine (Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, or Microsoft SQL Server) you want to use
`DBName`	Identifier for the database
`MasterUsername`	Name for the master user
`MasterUserPassword`	Password for the master user

It is possible to mark an RDS instance as publicly accessible. But we generally do not recommend you enable access from the internet to your database, to prevent unwanted

access. Instead, as shown in our example, an RDS instance should only be accessible within the VPC.

To connect to an RDS instance, you need an EC2 instance running in the same VPC. First, connect to the EC2 instance. From there, you can then connect to the RDS instance.

Listing 11.1 Extract from the CloudFormation template setting up an RDS database

```
Resources:
  # [...]
  DatabaseSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'awsinaction-db-sg'
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 3306
        ToPort: 3306
        SourceSecurityGroupId: !Ref WebServerSecurityGroup
  Database:
    Type: 'AWS::RDS::DBInstance'
    DeletionPolicy: Delete
    Properties:
      AllocatedStorage: 5
      BackupRetentionPeriod: 0
      DBInstanceClass: 'db.t2.micro'
      DBName: wordpress
      Engine: MySQL
      MasterUsername: wordpress
      MasterUserPassword: wordpress
      VPCSecurityGroups:
      - !Sub ${DatabaseSecurityGroup.GroupId}
      DBSubnetGroupName: !Ref DBSubnetGroup
    DependsOn: VPCGatewayAttachment
  DBSubnetGroup:
    Type: 'AWS::RDS::DBSubnetGroup'
    Properties:
      Description: DB subnet group
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
```

Security group for the database instance, allowing incoming traffic on the MySQL default port for web servers

The default MySQL port is 3306.

References the security group of EC2 instances running a web server

Creates a database instance with Amazon RDS

Disable backups (turn this to Snapshot in production!).

The database provides 5 GB of storage.

Disable backups (turn this on in production!).

Creates a default database named wordpress

The size of the database instance is t2.micro, the smallest available size.

Uses MySQL as the database engine

The password for the master user of the MySQL database

The username for the master user of the MySQL database

Defines the subnets the RDS database instance will launch into

References the security group for the database instance

Creates a subnet group to define the subnets for the database instance

Uses subnet A or B to launch RDS database instances

See if the CloudFormation stack named `wordpress` has reached the state `CREATE_COMPLETE` with the following command:

```
$ aws cloudformation describe-stacks --stack-name wordpress
```

Search for `StackStatus` in the output, and check whether the status is `CREATE_COMPLETE`. If not, you need to wait a few minutes longer (it can take up to 15 minutes to create the stack) and rerun the command. If the status is `CREATE_COMPLETE`, you'll find the key `OutputKey` in the output section. The corresponding `OutputValue` contains the URL for the WordPress blogging platform. The following listing shows the output in detail. Open this URL in your browser; you'll find a running WordPress setup.

Listing 11.2 Checking the state of the CloudFormation stack

```
$ aws cloudformation describe-stacks --stack-name wordpress
{
  "Stacks": [{
    "StackId": "[...]",
    "Description": "AWS in Action: chapter 11",
    "Parameters": [...],
    "Tags": [],
    "Outputs": [
        {
            "Description": "Wordpress URL",
            "OutputKey": "URL",
            "OutputValue": "http://[...].us-east-1.elb.amazonaws.com"
        }
    ],
    "CreationTime": "2017-10-19T07:12:28.694Z",
    "StackName": "wordpress",
    "NotificationARNs": [],
    "StackStatus": "CREATE_COMPLETE",
    "DisableRollback": false
  }]
}
```

Open this URL in your browser to open the WordPress application.

Wait for state CREATE_COMPLETE for the CloudFormation stack.

Launching and operating a relational database like MySQL is that simple. Of course, you can also use the Management Console (https://console.aws.amazon.com/rds/) to launch an RDS database instance instead of using a CloudFormation template. RDS is a managed service, and AWS handles most of the tasks necessary to operate your database in a secure and reliable way. You only need to do two things:

- Monitor your database's available storage and make sure you increase the allocated storage as needed.
- Monitor your database's performance and make sure you increase I/O and computing performance as needed.

Both tasks can be handled with the help of CloudWatch monitoring, as you'll learn later in the chapter.

11.1.2 Exploring an RDS database instance with a MySQL engine

The CloudFormation stack created an RDS database instance with a MySQL engine. Each instance offers an endpoint for SQL requests. Applications can send their SQL

requests to this endpoint to query, insert, delete, or update data. For example, to retrieve all rows from a table, the application sends the following SQL request: `SELECT * FROM table`. You can request the endpoint and detailed information of an RDS database instance with a `describe` command:

```
$ aws rds describe-db-instances --query "DBInstances[0].Endpoint"
{
  "HostedZoneId": "Z2R2ITUGPM61AM",
  "Port": 3306,
  "Address": "wdwcoq2o8digyr.cqrxioeaavmf.us-east-1.rds.amazonaws.com"
}
```

Port number of database endpoint

Host name of database endpoint

The RDS database is now running, but what does it cost?

11.1.3 Pricing for Amazon RDS

Databases on Amazon RDS are priced according to the size of the underlying virtual machine and the amount and type of allocated storage. Compared to a database running on a plain EC2 VM, the hourly price of an RDS instance is higher. In our opinion, the Amazon RDS service is worth the extra charge because you don't need to perform typical DBA tasks like installation, patching, upgrades, migration, backups, and recovery.

Table 11.3 shows a pricing example for a medium-sized RDS database instance without failover for high availability. All prices in USD are for US East (N. Virginia) as of Nov. 8, 2017. Get the current prices at https://aws.amazon.com/rds/pricing/.

Table 11.3 Monthly (30.5 days) cost for a medium-sized RDS instance

Description	Monthly price
Database instance db.m3.medium	$65.88 USD
50 GB of general purpose (SSD)	$5.75 USD
Additional storage for database snapshots (100 GB)	$9.50 USD
Total	$81.13 USD

You've now launched an RDS database instance for use with a WordPress web application. You'll learn about importing data to the RDS database in the next section.

11.2 Importing data into a database

A database without data isn't useful. In many cases, you'll need to import data into a new database, by importing a dump from the old database for example. If you move your locally hosted systems to AWS, you'll need to transfer the database as well. This section will guide you through the process of importing a MySQL database dump to an RDS database with a MySQL engine. The process is similar for all other database

engines (Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server).

To import a database from your local environment to Amazon RDS, follow these steps:

1. Export the local database.
2. Start a virtual machine in the same region and VPC as the RDS database.
3. Upload the database dump to the virtual machine.
4. Run an import of the database dump to the RDS database on the virtual server.

We'll skip the first step of exporting a MySQL database, because the RDS instance we created in our example is empty and you may not have access to an existing WordPress database. This sidebar gives you hints if you do.

Exporting a MySQL database

MySQL (and every other database system) offers a way to export and import databases. We recommend the command-line tools from MySQL for exporting and importing databases. You may need to install the MySQL client, which comes with the mysqldump tool.

The following command exports all databases from localhost and dumps them into a file called dump.sql. Replace *$UserName* with the MySQL admin or master user, and enter the password when prompted:

```
$ mysqldump -u $UserName -p --all-databases > dump.sql
```

You can also specify only some databases for the export. To do so, replace *$DatabaseName* with the name of the database you want to export:

```
$ mysqldump -u $UserName -p $DatabaseName > dump.sql
```

And of course you can export a database over a network connection. To connect to a server to export a database, replace *$Host* with the host name or IP address of your database:

```
$ mysqldump -u $UserName -p $DatabaseName --host $Host > dump.sql
```

See the MySQL documentation if you need more informations about the `mysqldump` tool.

Theoretically, you could import a database to RDS from any machine in your on-premises or local network. But the higher latency over the internet or VPN connection will slow down the import process dramatically. Because of this, we recommend adding a second step: upload the database dump to a virtual machine running in the same AWS region and VPC, and start to import the database to RDS from there.

> **AWS Database Migration Service**
> When migrating a huge database to AWS with minimal downtime, the Database Migration Service (DMS) can help. We do not cover DMS in this book, but you can learn more on the AWS website: https://aws.amazon.com/dms/.

To do so, we'll guide you through the following steps:

1. Get the public IP address of the virtual machine you want to upload to. We'll be using the VM running WordPress that you created earlier.
2. Connect to the virtual machine via SSH.
3. Download a database dump from S3 to the VM (or if you have an existing database dump that you're migrating, upload it to the VM).
4. Run an import of the database dump to the RDS database from the virtual machine.

Fortunately, you already started two virtual machines that you know can connect to the MySQL database on RDS, because they're running the WordPress application. To find out the public IP address of one of these two virtual machines, run the following command on your local machine:

```
$ aws ec2 describe-instances --filters "Name=tag-key,\
➥ Values=aws:cloudformation:stack-name Name=tag-value,\
➥ Values=wordpress" --output text \
➥ --query "Reservations[0].Instances[0].PublicIpAddress"
```

Open an SSH connection to the VM using the public IP address from the previous command. Use the SSH key `mykey` to authenticate, and replace *`$PublicIpAddress`* with the VM's IP address:

```
$ ssh -i $PathToKey/mykey.pem ec2-user@$PublicIpAddress
```

We prepared a MySQL database dump of a WordPress blog as an example. The dump contains a blog post and a few comments. Download this database dump from S3 using the following command on the virtual machine:

```
$ wget https://s3.amazonaws.com/awsinaction-code2/chapter11/wordpress-import.sql
```

Now you're ready to import the MySQL database dump to the RDS database instance. You'll need the port and hostname, also called the *endpoint*, of the MySQL database on RDS to do so. Don't remember the endpoint? The following command will print it out for you. Run this on your local machine:

```
$ aws rds describe-db-instances --query "DBInstances[0].Endpoint"
```

Run the following command on the VM to import the data from the file wordpress-import.sql into the RDS database instance; replace *$DBHostName* with the RDS endpoint you printed to the terminal with the previous command. Type in the password `wordpress` when asked for a password:

```
$ mysql --host $DBHostName --user wordpress -p < wordpress-import.sql
```

Point your browser to the WordPress blog again, and you'll now find many new posts and comments there. If you don't remember the URL, run the following command on your local machine to fetch it again:

```
$ aws cloudformation describe-stacks --stack-name wordpress \
➥ --query "Stacks[0].Outputs[0].OutputValue" --output text
```

11.3 Backing up and restoring your database

Amazon RDS is a managed service, but you still need backups of your database in case something or someone harms your data and you need to restore it, or you need to duplicate a database in the same or another region. RDS offers manual and automated snapshots for recovering RDS database instances.

In this section, you'll learn how to use RDS snapshots:

- Configuring the retention period and time frame for automated snapshots
- Creating snapshots manually
- Restoring snapshots by starting new database instances based on a snapshot
- Copying a snapshot to another region for disaster recovery or relocation

11.3.1 Configuring automated snapshots

The RDS database you started in section 11.1 can automatically create snapshots if the `BackupRetentionPeriod` is set to a value between `1` and `35`. This value indicates how many days the snapshot will be retained (default is `1`). Automated snapshots are created once a day during the specified time frame. If no time frame is specified, RDS picks a random 30-minute time frame during the night. (A new random time frame will be chosen each night.)

Creating a snapshot requires all disk activity to be briefly frozen. Requests to the database may be delayed or even fail because of a time out, so we recommend that you choose a time frame for the snapshot that has the least impact on applications and users (for example, late at night). Automated snapshots are your backup in case something unexpected happens to your database. This could be a query that deletes all your data accidentally or a hardware failure that causes data loss.

The following command changes the time frame for automated backups to 05:00–06:00 UTC and the retention period to three days. Use the terminal on your local machine to execute it:

```
$ aws cloudformation update-stack --stack-name wordpress --template-url \
➥ https://s3.amazonaws.com/awsinaction-code2/chapter11/ \
➥ template-snapshot.yaml \
➥ --parameters ParameterKey=KeyName,UsePreviousValue=true \
➥ ParameterKey=AdminPassword,UsePreviousValue=true \
➥ ParameterKey=AdminEMail,UsePreviousValue=true
```

The RDS database will be modified based on a slightly modified CloudFormation template, as shown next.

Listing 11.3 Modifying an RDS database's snapshot time frame and retention time

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.t2.micro'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
  DependsOn: VPCGatewayAttachment
```

Keep snapshots for 3 days.

Create snapshots automatically between 05:00 and 06:00 UTC.

If you want to disable automated snapshots, you need to set the retention period to `0`. As usual, you can configure automated backups using CloudFormation templates, the Management Console, or SDKs. Keep in mind that automated snapshots are deleted when the RDS database instance is deleted. Manual snapshots stay. You'll learn about them next.

11.3.2 Creating snapshots manually

You can trigger manual snapshots whenever you need, for example before you release a new version of your software, migrate a schema, or perform some other activity that could damage your database. To create a snapshot, you have to know the instance identifier. The following command extracts the instance identifier from the first RDS database instance:

```
$ aws rds describe-db-instances   --output text \
➥ --query "DBInstances[0].DBInstanceIdentifier"
```

The following command creates a manual snapshot called `wordpress-manual-snapshot`. Replace *`$DBInstanceIdentifier`* with the output of the previous command.

```
$ aws rds create-db-snapshot --db-snapshot-identifier \
➥ wordpress-manual-snapshot \
➥ --db-instance-identifier $DBInstanceIdentifier
```

It will take a few minutes for the snapshot to be created. You can check the current state of the snapshot with this command:

```
$ aws rds describe-db-snapshots \
➥ --db-snapshot-identifier wordpress-manual-snapshot
```

RDS doesn't delete manual snapshots automatically; you need to delete them yourself if you don't need them any longer. You'll learn how to do this at the end of the section.

Copying an automated snapshot as a manual snapshot

There is a difference between automated and manual snapshots. Automated snapshots are deleted automatically after the retention period is over, but manual snapshots aren't. If you want to keep an automated snapshot even after the retention period is over, you have to copy the automated snapshot to a new manual snapshot.

Get the snapshot identifier of an automated snapshot from the RDS database you started in section 11.1 by running the following command at your local terminal. Replace *$DBInstanceIdentifier* with the output of the `describe-db-instances` command.

```
$ aws rds describe-db-snapshots --snapshot-type automated \
➥ --db-instance-identifier $DBInstanceIdentifier \
➥ --query "DBSnapshots[0].DBSnapshotIdentifier" \
➥ --output text
```

The following command copies an automated snapshot to a manual snapshot named `wordpress-copy-snapshot`. Replace *$SnapshotId* with the output from the previous command:

```
$ aws rds copy-db-snapshot \
➥ --source-db-snapshot-identifier $SnapshotId  \
➥ --target-db-snapshot-identifier wordpress-copy-snapshot
```

The copy of the automated snapshot is named `wordpress-copy-snapshot`. It won't be removed automatically.

11.3.3 Restoring a database

If you restore a database from an automated or manual snapshot, a new database will be created based on the snapshot. As figure 11.2 shows, you can't restore a snapshot to an existing database.

A new database is created when you restore a database snapshot, as figure 11.3 illustrates.

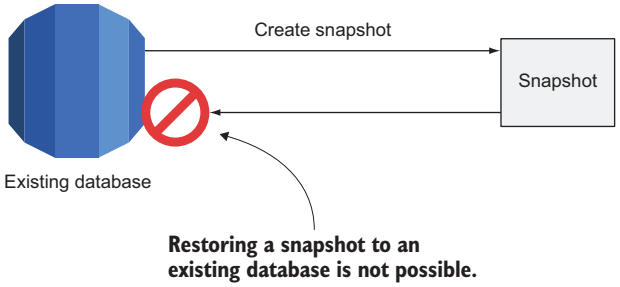


Figure 11.2 A snapshot can't be restored into an existing database.

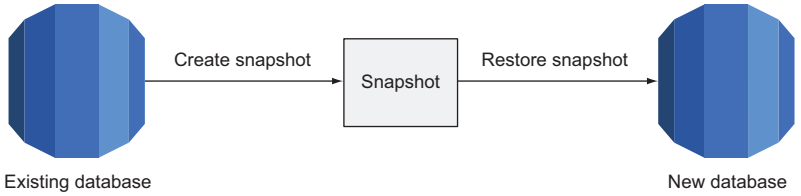


Figure 11.3 A new database is created to restore a snapshot.

To create a new database in the same VPC as the WordPress platform you started in section 11.1, you need to find out the existing database's subnet group. Execute this command to do so:

```
$ aws cloudformation describe-stack-resource \
➥ --stack-name wordpress --logical-resource-id DBSubnetGroup \
➥ --query "StackResourceDetail.PhysicalResourceId" --output text
```

You're now ready to create a new database based on the manual snapshot you created at the beginning of this section. Execute the following command, replacing *$Subnet-Group* with the output of the previous command:

```
$ aws rds restore-db-instance-from-db-snapshot \
➥ --db-instance-identifier awsinaction-db-restore \
➥ --db-snapshot-identifier wordpress-manual-snapshot \
➥ --db-subnet-group-name $SubnetGroup
```

A new database named `awsinaction-db-restore` is created based on the manual snapshot. After the database is created, you can switch the WordPress application to the new endpoint.

If you're using automated snapshots, you can also restore your database from a specified moment, because RDS keeps the database's change logs. This allows you to

jump back to any point in time from the backup retention period to the last five minutes.

Execute the following command, replacing *$DBInstanceIdentifier* with the output of the earlier `describe-db-instances` command, *$SubnetGroup* with the output of the earlier `describe-stack-resource` command, and *$Time* with a UTC timestamp from 5 minutes ago (for example, `2017-10-19T10:55:00Z`):

```
$ aws rds restore-db-instance-to-point-in-time \
➥ --target-db-instance-identifier awsinaction-db-restore-time \
➥ --source-db-instance-identifier $DBInstanceIdentifier \
➥ --restore-time $Time --db-subnet-group-name $SubnetGroup
```

A new database named `awsinaction-db-restore-time` is created based on the source database from 5 minutes ago. After the database is created, you can switch the WordPress application to the new endpoint.

11.3.4 Copying a database to another region

Copying a database to another region is easy with the help of snapshots. The main reasons you might do so are:

- *Disaster recovery*—You can recover from an unlikely region-wide outage.
- *Relocating*—You can move your infrastructure to another region so you can serve your customers with lower latency.

You can easily copy a snapshot to another region. The following command copies the snapshot named `wordpress-manual-snapshot` from the region `us-east-1` to the region `eu-west-1`. You need to replace *$AccountId* with your account ID.

> **COMPLIANCE** Moving data from one region to another may violate privacy laws or compliance rules, especially if the data crosses frontiers. Make sure you're allowed to copy the data to another region if you're working with real data.

```
$ aws rds copy-db-snapshot --source-db-snapshot-identifier \
➥ arn:aws:rds:us-east-1:$AccountId:snapshot:\
➥ wordpress-manual-snapshot --target-db-snapshot-identifier \
➥ wordpress-manual-snapshot --region eu-west-1
```

If you can't remember your account ID, you can look it up with the help of the CLI:

```
$ aws iam get-user --query "User.Arn" --output text
arn:aws:iam::878533158213:user/mycli
```

Account ID has 12 digits (878533158213).

After the snapshot has been copied to the region `eu-west-1`, you can restore a database from it as described in the previous section.

11.3.5 Calculating the cost of snapshots

Snapshots are billed based on the storage they use. You can store snapshots up to the size of your database instance for free. In our WordPress example, you can store up to 5 GB of snapshots for free. On top of that, you pay per GB per month of used storage. As we're writing this book, the cost is $0.095 for each GB every month (region `us-east-1`).

Cleaning up

It's time to clean up the snapshots and delete the restored database instances. Execute the following commands step-by-step, or jump to the shortcuts for Linux and macOS after the listing:

```
$ aws rds delete-db-instance --db-instance-identifier \
➥ awsinaction-db-restore --skip-final-snapshot
$ aws rds delete-db-instance --db-instance-identifier \
➥ awsinaction-db-restore-time --skip-final-snapshot
$ aws rds delete-db-snapshot --db-snapshot-identifier \
➥ wordpress-manual-snapshot
$ aws rds delete-db-snapshot --db-snapshot-identifier \
➥ wordpress-copy-snapshot
$ aws --region eu-west-1 rds delete-db-snapshot --db-snapshot-identifier \
➥ wordpress-manual-snapshot
```

Deletes the database with data from the snapshot restore

Deletes the database with data from the point-in-time restore

Deletes the manual snapshot

Deletes the copied snapshot

Deletes the snapshot copied to another region

You can avoid typing these commands manually at your terminal by using the following command to download a Bash script and execute it directly on your local machine. The Bash script contains the same steps as shown in the previous snippet.

```
$ curl -s https://raw.githubusercontent.com/AWSinAction/\
➥ code2/master/chapter11/cleanup.sh | bash -ex
```

Keep the rest of the setup, because you'll use it in the following sections.

11.4 Controlling access to a database

The shared-responsibility model applies to the RDS service as well as to AWS services in general. AWS is responsible for security of the cloud in this case—for example, for the security of the underlying OS. You, the customer, need to specify the rules controlling access to your data and RDS database.

Figure 11.4 shows the three layers that control access to an RDS database:

1. Controlling access to the configuration of the RDS database
2. Controlling network access to the RDS database
3. Controlling data access with the database's own user and access management features

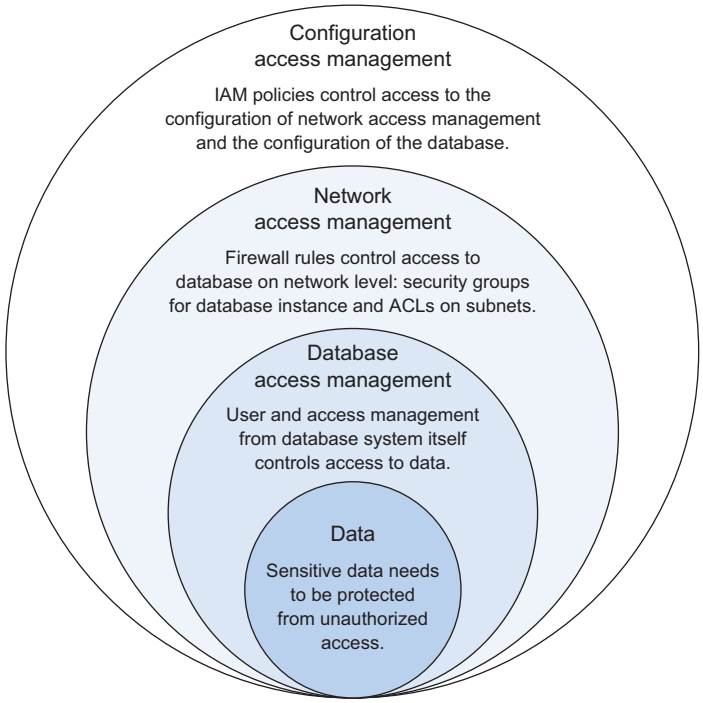


Figure 11.4 Your data is protected by the database itself, security groups, and IAM.

11.4.1 Controlling access to the configuration of an RDS database

Access to the RDS service is controlled using the IAM service. IAM is responsible for controlling access to actions like creating, updating, and deleting an RDS database instance. IAM doesn't manage access inside the database; that's the job of the database engine. IAM policies define which configuration and management actions an identity is allowed to execute on RDS. You attach these policies to IAM users, groups, or roles to control what actions they can perform on the database.

The following listing shows an IAM policy that allows access to all RDS configuration and management actions. You could use this policy to limit access by only attaching it to trusted IAM users and groups.\

Listing 11.4 Allowing access to all RDS service configuration and management actions

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Stmt1433661637000",
    "Effect": "Allow",
    "Action": "rds:*",
    "Resource": "*"
  }]
}
```

Allows the specified actions on the specified resources

All possible actions on RDS service are specified (for example, changes to the database configuration).

All RDS databases are specified.

Only people and machines that really need to make changes to RDS databases should be allowed to do so. The following listing shows an IAM policy that denies all destructive actions in order to prevent data loss by human failure.

Listing 11.5 IAM policy denying destructive actions

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "Stmt1433661637000",
    "Effect": "Deny",
    "Action": ["rds:Delete*", "rds:Remove*"],
    "Resource": "*"
  }]
}
```

Denies the specified actions on the specified resources

All destructive actions on the RDS service are specified (e.g., delete database instance).

All RDS databases are specified.

See chapter 6 if you're interested in more details about the IAM service.

11.4.2 Controlling network access to an RDS database

An RDS database is linked to security groups. Each security group consists of rules for a firewall controlling inbound and outbound database traffic. You already know about using security groups in combination with virtual machines.

The next listing shows the configuration of the security group attached to the RDS database in our WordPress example. Inbound connections to port 3306 (the default port for MySQL) are only allowed from virtual machines linked to the security group called `WebServerSecurityGroup`.

Listing 11.6 CloudFormation template extract: firewall rules for an RDS database

```
DatabaseSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'awsinaction-db-sg'
    VpcId: !Ref VPC
    SecurityGroupIngress:
    - IpProtocol: tcp
      FromPort: 3306
      ToPort: 3306
      SourceSecurityGroupId: !Ref WebServerSecurityGroup
```

Security group for the database instance, allowing incoming traffic on the MySQL default port for web servers

The default MySQL port is 3306.

References the security group for web servers

Only machines that really need to connect to the RDS database should be allowed to do so on the network level, such as EC2 instances running your web server or application server. See chapter 6 if you're interested in more details about security groups (firewall rules).

11.4.3 Controlling data access

A database engine also implements access control itself. User management of the database engine has nothing to do with IAM users and access rights; it's only responsible for controlling access to the database. For example, you typically define a user for each application and grant rights to access and manipulate tables as needed. In the WordPress example, a database user called wordpress is created. The WordPress application authenticates itself to the database engine (MySQL in this case) with this database user and a password.

> **IAM Database Authentication**
>
> AWS has started integrating IAM and the native database authentication mechanism for two engines: MySQL and Aurora. With IAM Database Authentication, you no longer need to create users with a username and password in the database engine. Instead, you create a database user that uses a plugin called `AWSAuthenticationPlugin` for authentication. You then log in to the database with the username and a token that is generated with your IAM identity. The token is valid for 15 minutes, so you have to renew it from time to time. You can learn more about IAM Database Authentication in the AWS documentation at http://mng.bz/5q65.

Typical use cases are as follows:

- Limiting write access to a few database users (for example, only for an application)
- Limiting access to specific tables to a few users (for example, to one department in the organization)
- Limiting access to tables to isolate different applications (for example, hosting multiple applications for different customers on the same database)

User and access management varies between database systems. We don't cover this topic in this book; refer to your database system's documentation for details.

11.5 Relying on a highly available database

The database is typically the most important part of a system. Applications won't work if they can't connect to the database, and the data stored in the database is mission-critical, so the database must be highly available and store data durably.

Amazon RDS lets you launch highly available (HA) databases. Compared to a default database consisting of a single database instance, an HA RDS database consists of two database instances: a master and a standby database. You also pay for both instances. All clients send requests to the master database. Data is replicated between the master and the standby database synchronously, as shown in figure 11.5.

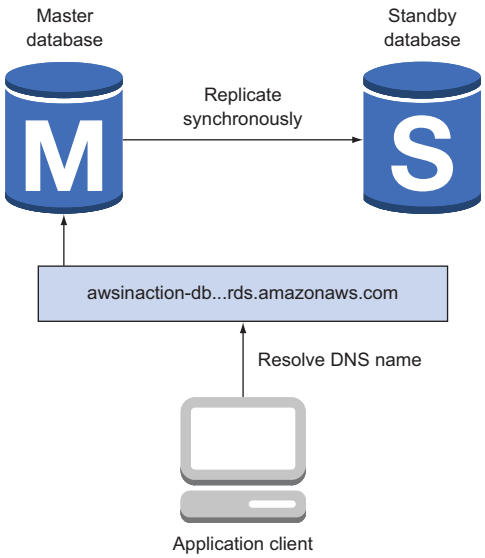


Figure 11.5 The master database is replicated to the standby database when running in high-availability mode.

We strongly recommend using high-availability deployment for all databases that handle production workloads. If you want to save money, you can turn the HA feature off for your test systems.

If the master database becomes unavailable due to hardware or network failures, RDS starts the failover process. The standby database then becomes the master database. As figure 11.6 shows, the DNS name is updated and clients begin to use the former standby database for their requests.

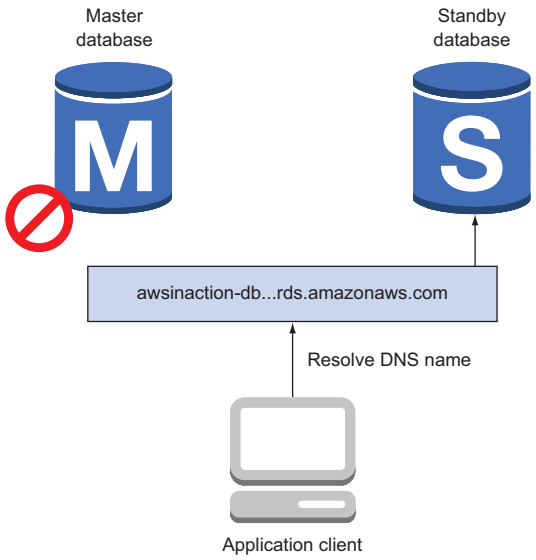


Figure 11.6 The client fails over to the standby database if the master database fails, using DNS resolution.

RDS detects the need for a failover automatically and executes it without human intervention.

> **Aurora is different**
>
> Aurora is an exception. It does not store your data on a single EBS volume. Instead, Aurora stores data on a *cluster volume*. A cluster volume consists of multiple disks, with each disk having a copy of the cluster data. This implies that the storage layer of Aurora is not a single point of failure. But still, only the primary Aurora database instance accepts write requests. If the primary goes down, it is automatically re-created, which typically takes less than 10 minutes. If you have replica instances in your Aurora cluster, a replica is promoted to be the new primary instance, which usually takes around 1 minute and is much faster than primary re-creation.

11.5.1 Enabling high-availability deployment for an RDS database

Execute the following command at your local terminal to enable high-availability deployment for the RDS database you started in section 11.1:

```
$ aws cloudformation update-stack --stack-name wordpress --template-url \
➥ https://s3.amazonaws.com/awsinaction-code2/chapter11/template-multiaz.yaml \
➥ --parameters ParameterKey=KeyName,UsePreviousValue=true \
➥ ParameterKey=AdminPassword,UsePreviousValue=true \
➥ ParameterKey=AdminEMail,UsePreviousValue=true
```

> **WARNING** Starting a highly available RDS database will incur charges. See https://aws.amazon.com/rds/pricing/ if you want to find out the current hourly price.

The RDS database is updated based on a slightly modified CloudFormation template.

Listing 11.7 Modifying the RDS database by enabling high availability

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.t2.micro'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
    MultiAZ: true
  DependsOn: VPCGatewayAttachment
```

Enables high-availability deployment for the RDS database

It will take several minutes for the database to be deployed in HA mode. But there is nothing more you need to do—the database is now highly available.

> **What is Multi-AZ ?**
> Each AWS region is split into multiple independent data centers, also called *availability zones*. We introduced the concept of availability zones in chapters 9 and 10, but skipped one aspect of HA deployment that is only used for RDS: the master and standby databases are launched into two different availability zones. AWS calls the high-availability deployment of RDS *Multi-AZ* deployment for this reason.

In addition to the fact that a high-availability deployment increases your database's reliability, there is another important advantage. Reconfiguring or maintaining a single-mode database causes short downtimes. High-availability deployment of an RDS database solves this problem because you can switch to the standby database during maintenance.

11.6 Tweaking database performance

The easiest way to scale a RDS database, or a SQL database in general, is to scale *vertically*. Scaling a database vertically means increasing the resources of your database instance:

- Faster CPU
- More memory
- Faster storage

Keep in mind that you can't scale vertically (which means increasing resources) without limits. One of the largest RDS database instance types comes with 32 cores and 244 GiB memory. In comparison, an object store like S3 or a NoSQL database like DynamoDB can be scaled horizontally without limits, as they add more machines to the cluster if additional resources are needed.

11.6.1 Increasing database resources

When you start an RDS database, you choose an instance type. The instance type defines the computing power and memory of your virtual machine (as when you start an EC2 instance). Choosing a bigger instance type increases computing power and memory for RDS databases.

You started an RDS database with instance type `db.t2.micro`, the smallest available instance type. You can change the instance type using a CloudFormation template, the CLI, the Management Console, or AWS SDKs. You may want to increase the instance type if performance is not good enough for you. You will learn how to measure performance in section 11.7. Listing 11.8 shows how to change the CloudFormation template to increase the instance type from `db.t2.micro` with 1 virtual core and 615 MB memory to `db.m3.large` with 2 faster virtual cores and 7.5 GB memory. You'll

do this only in theory. Don't do this to your running database because it is not covered by the Free Tier and will incur charges.

Listing 11.8 Modifying the instance type to improve performance of an RDS database

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
    DBInstanceClass: 'db.m3.large'
    DBName: wordpress
    Engine: MySQL
    MasterUsername: wordpress
    MasterUserPassword: wordpress
    VPCSecurityGroups:
    - !Sub ${DatabaseSecurityGroup.GroupId}
    DBSubnetGroupName: !Ref DBSubnetGroup
    MultiAZ: true
  DependsOn: VPCGatewayAttachment
```

Increases the size of the underlying virtual machine for the database instance from db.t2.micro to db.m3.large

Because a database has to read and write data to a disk, I/O performance is important for the database's overall performance. RDS offers three different types of storage, as you already know from reading about the block storage service EBS:

1. General purpose (SSD)
2. Provisioned IOPS (SSD)
3. Magnetic

You should choose general purpose (SSD) or even provisioned IOPS (SSD) storage for production workloads. The options are exactly the same as when using EBS for virtual machines. If you need to guarantee a high level of read or write throughput, you should use provisioned IOPS (SSD). The general purpose (SSD) option offers moderate baseline performance with the ability to burst. The throughput for general purpose (SSD) depends on the amount of initialized storage size. Magnetic storage is an option if you need to store data at a low cost, or if you don't need to access it in a predictable, performant way. The next listing shows how to enable general purpose (SSD) storage using a CloudFormation template.

Listing 11.9 Modifying the storage type to improve performance of an RDS database

```
Database:
  Type: 'AWS::RDS::DBInstance'
  DeletionPolicy: Delete
  Properties:
    AllocatedStorage: 5
    BackupRetentionPeriod: 3
    PreferredBackupWindow: '05:00-06:00'
```

```
      DBInstanceClass: 'db.m3.large'
      DBName: wordpress
      Engine: MySQL
      MasterUsername: wordpress
      MasterUserPassword: wordpress
      VPCSecurityGroups:
      - !Sub ${DatabaseSecurityGroup.GroupId}
      DBSubnetGroupName: !Ref DBSubnetGroup
      MultiAZ: true
      StorageType: 'gp2'                    <-- Uses general purpose (SSD) storage
    DependsOn: VPCGatewayAttachment             to increase I/O performance
```

11.6.2 Using read replication to increase read performance

A database suffering from too many read requests can be scaled horizontally by adding additional database instances for read replication. As figure 11.7 shows, changes to the database are asynchronously replicated to an additional read-only database instance. The read requests can be distributed between the master database and its read-replication databases to increase read throughput.

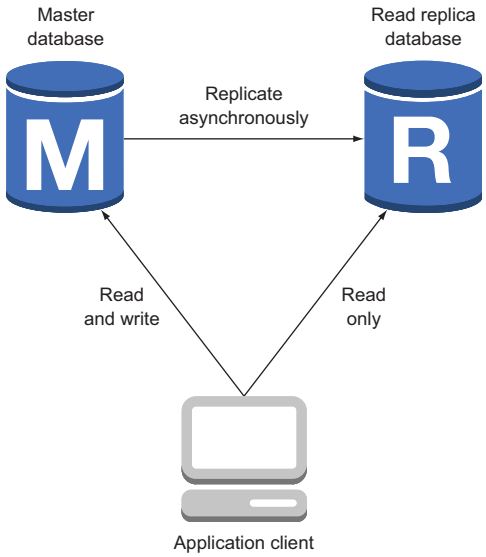


Figure 11.7 Read requests are distributed between the master and read-replication databases for higher read performance.

Tweaking read performance with replication makes sense only if the application generates many read requests and few write requests. Fortunately, most applications read more than they write.

CREATING A READ-REPLICATION DATABASE

Amazon RDS supports read replication for MySQL, MariaDB, and PostgreSQL databases. To use read replication, you need to enable automatic backups for your database, as shown in section 11.3.

WARNING Starting an RDS read replica will incur charges. See https://aws.amazon.com/rds/pricing/ if you want to find out the current hourly price.

Execute the following command from your local machine to create a read-replication database for the WordPress database you started in section 11.1. Replace the *$DBInstanceIdentifier* with the value from `aws rds describe-db-instances --query "DBInstances[0].DBInstanceIdentifier" --output text`.

```
$ aws rds create-db-instance-read-replica \
➥ --db-instance-identifier awsinaction-db-read \
➥ --source-db-instance-identifier $DBInstanceIdentifier
```

RDS automatically triggers the following steps in the background:

1. Creating a snapshot from the source database, also called the master database
2. Launching a new database based on that snapshot
3. Activating replication between the master and read-replication databases
4. Creating an endpoint for SQL read requests to the read-replication database

After the read-replication database is successfully created, it's available to answer SQL read requests. The application using the SQL database must support the use of read-replication databases. WordPress, for example, doesn't support read replicas by default, but you can use a plugin called HyperDB to do so; the configuration is tricky, so we'll skip this part. You can get more information here: https://wordpress.org/plugins/hyperdb/. Creating or deleting a read replica doesn't affect the availability of the master database.

Using read replication to transfer data to another region

RDS supports read replication between regions for Aurora, MySQL, MariaDB, and PostgreSQL databases. You can replicate your data from the data centers in North Virginia to the data centers in Ireland, for example. There are three major use cases for this feature:

1. Backing up data to another region for the unlikely event of an outage covering a complete region
2. Transferring data to another region to be able to answer read requests with lower latency
3. Migrating a database to another region

Creating read replication between two regions incurs an additional cost because you have to pay for the transferred data.

Promoting a read replica to a standalone database

If you create a read-replication database to migrate a database from one region to another, or if you have to perform heavy and load-intensive tasks on your database, such as adding an index, it's helpful to switch your workload from the master database

to a read-replication database. The read replica must become the new master database. Promoting read-replication databases to become master databases is possible for Aurora, MySQL, MariaDB, and PostgreSQL databases with RDS.

The following command promotes the read-replication database you created in this section to a standalone master database. Note that the read-replication database will perform a restart and be unavailable for a few minutes:

```
$ aws rds promote-read-replica --db-instance-identifier awsinaction-db-read
```

The RDS database instance named `awsinaction-db-read` will accept write requests after the transformation is successful.

Cleaning up

It's time to clean up, to avoid unwanted expense. Execute the following command:

```
$ aws rds delete-db-instance --db-instance-identifier \
➥ awsinaction-db-read --skip-final-snapshot
```

You've gained experience with the AWS relational database service in this chapter. We'll end the chapter by taking a closer look at the monitoring capabilities of RDS.

11.7 Monitoring a database

RDS is a managed service. Nevertheless, you need to monitor some metrics yourself to make sure your database can respond to all requests from applications. RDS publishes several metrics for free to AWS CloudWatch, a monitoring service for the AWS cloud. You can watch these metrics through the Management Console, as shown in figure 11.8, and define alarms for when a metric reaches a threshold.

Table 11.4 shows the most important metrics; we recommend that you keep an eye on them by creating alarms.

Table 11.4 Important metrics for RDS databases from CloudWatch

Name	Description
`FreeStorageSpace`	Available storage in bytes. Make sure you don't run out of storage space. We recommend setting the alarm threshold to < 2147483648 (2 GB)
`CPUUtilization`	The usage of the CPU as a percentage. High utilization can be an indicator of a bottleneck due to insufficient CPU performance. We recommend setting the alarm threshold to > 80%.
`FreeableMemory`	Free memory in bytes. Running out of memory can cause performance problems. We recommend setting the alarm threshold to < 67108864 (64 MB).

Table 11.4 Important metrics for RDS databases from CloudWatch *(continued)*

Name	Description
`DiskQueueDepth`	Number of outstanding requests to the disk. A long queue indicates that the database has reached the storage's maximum I/O performance. We recommend setting the alarm threshold to > 64.
`SwapUsage`	If the database has insufficient memory, the OS starts to use the disk as memory (this is called *swapping*). Using the disk as memory is slow and will cause performance issues. We recommend setting the alarm threshold to > 268435456 (256 MB).

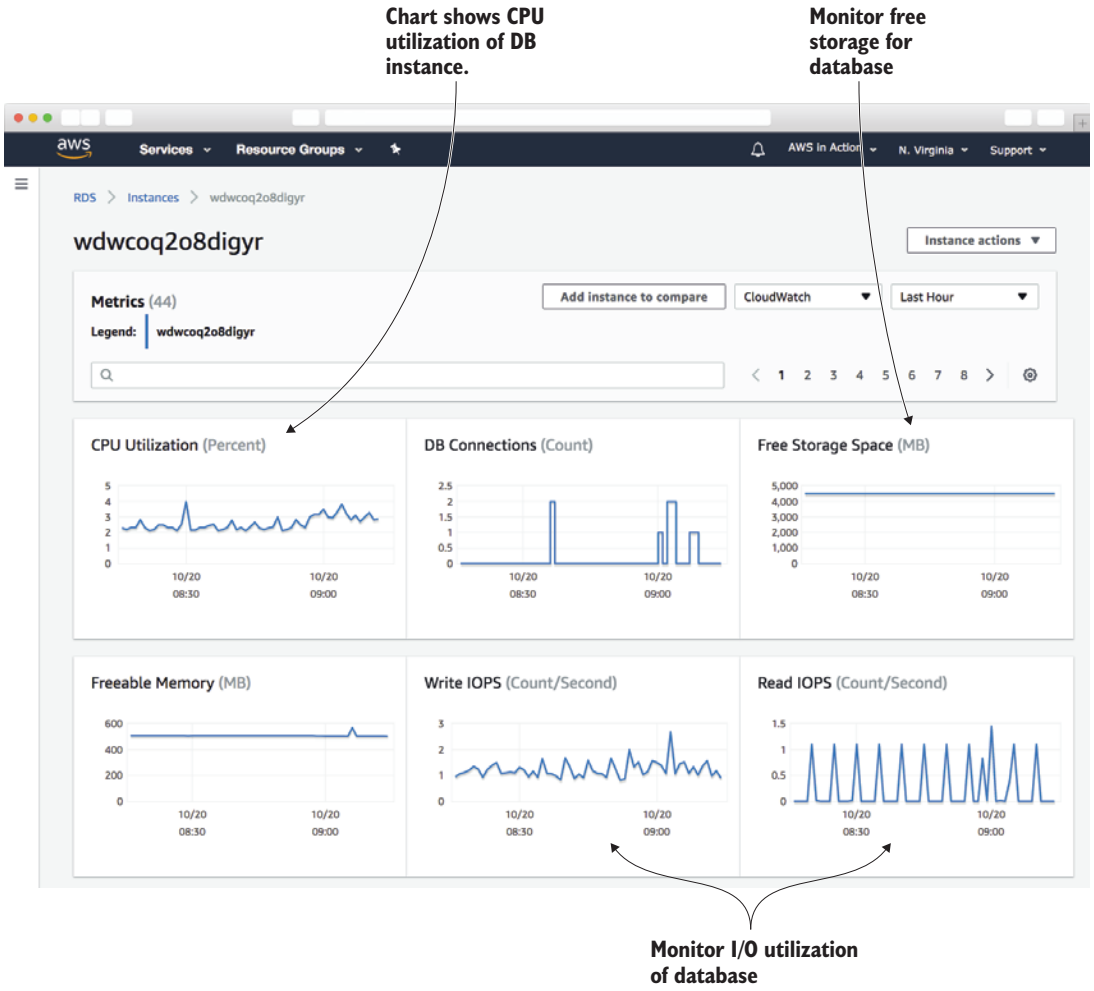


Figure 11.8 Metrics to monitor an RDS database from the Management Console

We recommend that you monitor these metrics in particular, to make sure your database isn't the cause of application performance problems.

Cleaning up

It's time to clean up, to avoid unwanted expense. Execute the following command to delete all resources corresponding to the WordPress blogging platform based on an RDS database:

```
$ aws cloudformation delete-stack --stack-name wordpress
```

In this chapter, you've learned how to use the RDS service to manage relational databases for your applications. The next chapter will focus on a NoSQL database.

Summary

- RDS is a managed service that provides relational databases.
- You can choose between PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server databases. Aurora is the database engine built by Amazon.
- The fastest way to import data into an RDS database is to copy it to a virtual machine in the same region and pump it into the RDS database from there.
- You can control access to data with a combination of IAM policies and firewall rules, and on the database level.
- You can restore an RDS database to any time in the retention period (a maximum of 35 days).
- RDS databases can be highly available. You should launch RDS databases in Multi-AZ mode for production workloads.
- Read replication can improve the performance of read-intensive workloads on a SQL database.

12 Caching data in memory: Amazon ElastiCache

This chapter covers

- Benefits of a caching layer between your application and data store
- Terminology like cache cluster, node, shard, replication group, and node group
- Using/Operating an in-memory key-value store
- Performance tweaking and monitoring ElastiCache clusters

Imagine a relational database being used for a popular mobile game where players' scores and ranks are updated and read frequently. The read and write pressure to the database will be extremely high, especially when ranking scores across millions of players. Mitigating that pressure by scaling the database may help with load, but not necessarily the latency or cost. Also, relational databases tend to be more expensive than caching data stores.

A proven solution used by many gaming companies is leveraging an in-memory data store such as Redis for both caching and ranking player and game metadata.

Instead of reading and sorting the leaderboard directly from the relational database, they store an in-memory game leaderboard in Redis, commonly using a Redis Sorted Set, which will sort the data automatically when it's inserted based on the score parameter. The score value may consist of the actual player ranking or player score in the game.

Because the data resides in memory and does not require heavy computation to sort, retrieving the information is incredibly fast, leaving little reason to query from a relational database. In addition, any other game and player metadata such as player profile, game level information, and so on that requires heavy reads can also be cached within this in-memory layer, freeing the database from heavy read traffic.

In this solution, both the relational database and in-memory layer will store updates to the leaderboard: one will serve as the primary database and the other as the working and fast processing layer. For caching data, they may employ a variety of caching techniques to keep the data that's cached fresh, which we'll review later. Figure 12.1 shows where the cache sits between your application and the database.

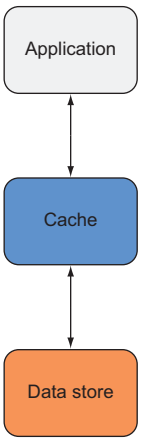


Figure 12.1 Cache sits between the application and the database

A cache comes with multiple benefits:

- The read traffic can be served from the caching layer, which frees resources on your data store, for example for write requests.
- It speeds up your application because the caching layer responds more quickly than your data store.
- You can downsize your data store, which can be more expensive than the caching layer.

Most caching layers reside in-memory and that's why they are so fast. The downside is that you can lose the cached data at any time because of a hardware defect or a restart. Always keep a copy of your data in a primary data store with disk durability, like the relational database in the mobile game example. Alternatively, Redis has optional failover support. In the event of a node failure, a replica node will be elected to be the new primary and will already have a copy of the data.

Depending on your caching strategy, you can either populate the cache in real-time or on-demand. In the mobile game example, on-demand means that if the leaderboard is not in the cache, the application asks the relational database and puts the result into the cache. Any subsequent request to the cache will result in a cache hit, meaning the data is found. This will be true until the duration of the TTL (time to live) value on the cached value expires. This strategy is called *lazy-loading* the data from the primary data store. Additionally, we could have a cron job running in the background that queries the leaderboard from the relational database every minute and puts the result in the cache to populate the cache in advance.

The lazy-loading strategy (getting data on demand) is implemented like this:

1. The application writes data to the data store.
2. If the application wants to read the data, at a later time it makes a request to the caching layer.
3. The caching layer does not contain the data. The application reads from the data store directly and puts the read value into the cache, and also returns the value to the client.
4. Later, if the application wants to read the data again, it makes a request to the caching layer and finds the value.

This strategy comes with a problem. What if the data is changed while it is in the cache? The cache will still contain the old value. That's why setting an appropriate TTL value is critical to ensure cache validity. Let's say you apply a TTL of 5 minutes to your cached data: this means you accept that the data could be up to 5 minutes out of sync with your primary database. Understanding the frequency of change for the underlying data and the effects out-of-sync data will have on the user experience is the first step of identifying the appropriate TTL value to apply. A common mistake some developers make is assuming that a few seconds of a cache TTL means that having a cache is not worthwhile. Remember that within those few seconds, millions of requests can be eliminated from your back end, speeding up your application and reducing the back-end database pressure. Performance testing your application with and without your cache, along with various caching approaches, will help fine-tune your implementation. In summary, the shorter the TTL, the more load you have on your underlying data store. The higher the TTL, the more out of sync the data gets.

The write-through strategy (caching data up front) is implemented differently to tackle the synchronization issue:

1. The application writes data to the data store and the cache (or the cache is filled asynchronously, for example in a cron job, AWS Lambda function, or the application).
2. If the application wants to read the data at a later time, it makes a request to the caching layer, which contains the data.
3. The value is returned to the client.

This strategy also comes with a problem. What if the cache is not big enough to contain all your data? Caches are in-memory and your data store's disk capacity is usually larger than your cache's memory capacity. When your cache reaches the available memory, it will evict data, or stop accepting new data. In both situations, the application stops working. In the gaming app, the global leaderboard will always fit into the cache. Imagine that a leaderboard is 4 KB in size and the cache has a capacity of 1 GB (1,048,576 KB). But what about team leaderboards? You can only store 262,144 (1,048,576 / 4) leaderboards, so if you have more teams than that, you will run into an capacity issue.

Figure 12.2 compares the two caching strategies. When evicting data, the cache needs to decide which data it should delete. One popular strategy is to evict the least recently used (LRU) data. This means that cached data must contain meta information about the time when it was last accessed. In case of an LRU eviction, the data with the oldest timestamp is chosen for eviction.

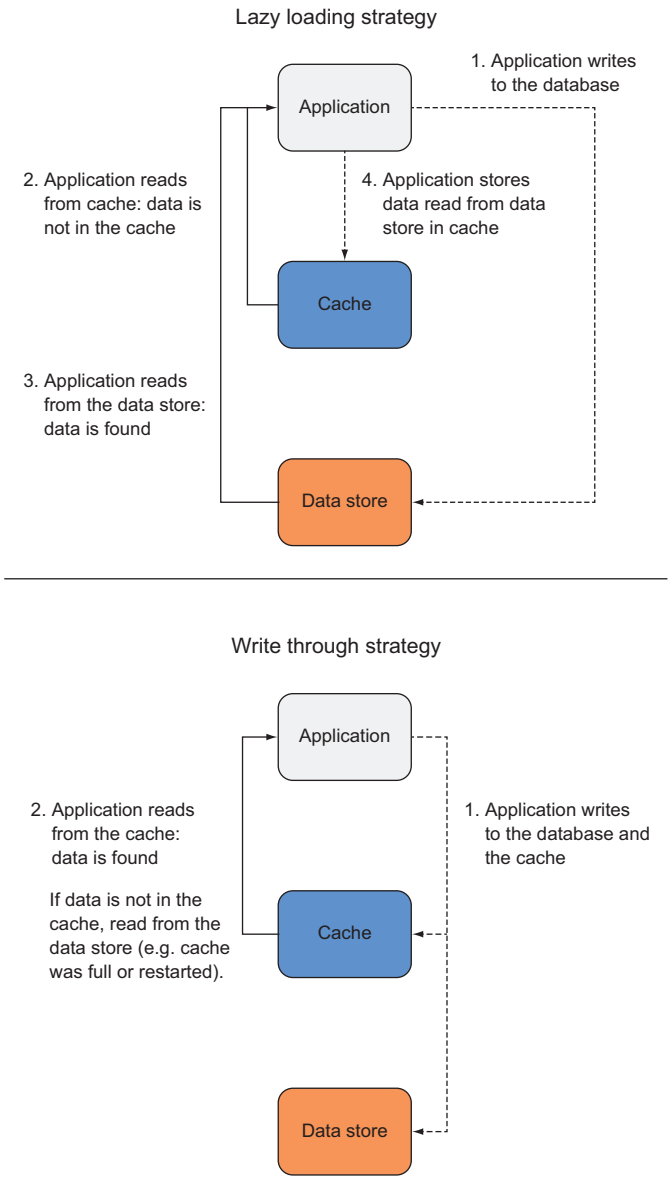


Figure 12.2 Comparing the lazy-loading and write-through caching strategies

Caches are usually implemented using key-value stores. Key-value stores don't support sophisticated query languages such as SQL. They support retrieving data based on a key, usually a string, or specialized commands, for example to extract sorted data efficiently.

Imagine that in your relational database you have a player table for your mobile game. One of the most common queries is `SELECT id, nickname FROM player ORDER BY score DESC LIMIT 10` to retrieve the top ten players. Luckily, the game is very popular. But this comes with a technical challenge. If many players look at the leaderboard, the database becomes very busy, which causes high latency or even time-outs. You have to come up with a plan to reduce the load on the database. As you already learned, caching can help. What technique should you employ for caching? You have a few options.

One approach you can take with Redis is to store the result of your SQL query as a String value and the SQL statement as your key name. Instead of using the whole SQL query as the key, you can hash the string with a hash function like md5 or sha256 to optimize storage and bandwidth ❶ as shown in figure 12.3. Before the application sends the query to the database, it takes the SQL query as the key to ask the caching layer for data ❷. If the cache does not contain data for the key ❸, the SQL query is sent to the relational database ❹. The result ❺ is then stored in the cache using the SQL query as the key ❻. The next time the application wants to perform the query, it asks the caching layer ❼, which now contains the cached table ❽.

To implement caching, you only need to know the key of the cached item. This can be an SQL query, a filename, a URL, or a user ID. You take the key and ask the cache for a result. If no result is found, you make a second call to the underlying data store, which knows the truth.

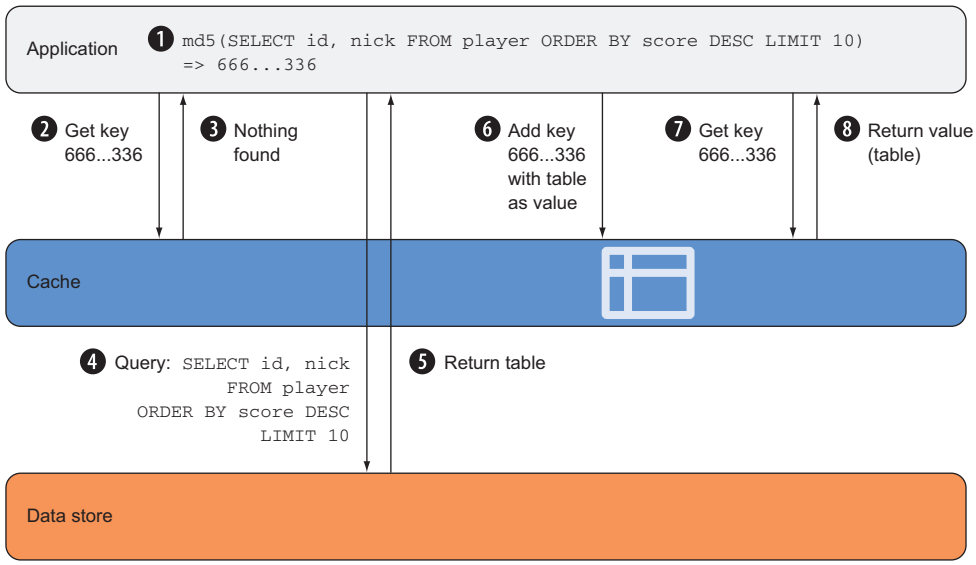


Figure 12.3 SQL caching layer implementation

With Redis, you also have the option of storing the data in other data structures such as a Redis SortedSet. If the data is stored in a Redis SortedSet, retrieving the ranked data will be very efficient. You could simply store players and their scores and sort by the score. An equivalent SQL command would be:

```
ZREVRANGE "player-scores" 0 9
```

This would return the ten players in a SortedSet named "player-scores" ordered from highest to lowest.

> **Examples are 100% covered by the Free Tier**
>
> The examples in this chapter are completely covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything. Keep in mind that this only applies if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

The two most popular implementations of in-memory key-value stores are Memcached and Redis. Amazon ElastiCache offers both options. Table 12.1 compares their features.

Table 12.1 Comparing Memcached and Redis features

	Memcached	Redis
Data types	simple	complex
Data manipulation commands	12	125
Server-side scripting	no	yes (Lua)
Transactions	no	yes
Multi-threaded	yes	no

Amazon ElastiCache offers Memcached and Redis clusters as a service. Therefore, AWS covers the following aspects for you:

- *Installation*—AWS installs the software for you and has enhanced the underlying engines.
- *Administration*—AWS administers Memcached/Redis for you and provides ways to configure your cluster through parameter groups. AWS also detects and automates failovers (Redis only).
- *Monitoring*—AWS publishes metrics to CloudWatch for you.
- *Patching*—AWS performs security upgrades in a customizable time window.

- *Backups*—AWS optionally backs up your data in a customizable time window (Redis only).
- *Replication*—AWS optionally sets up replication (Redis only).

Next, you will learn how to create an in-memory cluster with ElastiCache that you will later use as an in-memory cache for an application.

12.1 Creating a cache cluster

In this chapter, we focus on the Redis engine because it's more flexible. You can choose which engine to use based on the features that we compared in the previous section. If there are significant differences to Memcached, we will highlight them.

12.1.1 Minimal CloudFormation template

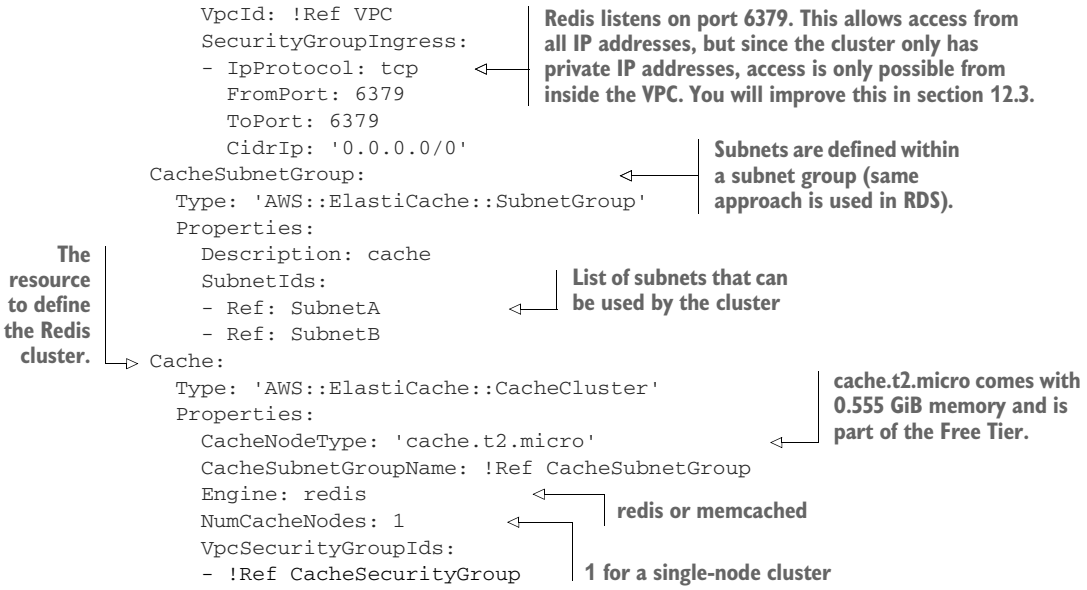
You can create an ElastiCache cluster using the Management Console, the CLI, or CloudFormation. You will use CloudFormation in this chapter to manage your cluster. The resource type of an ElastiCache cluster is `AWS::ElastiCache::CacheCluster`. The required properties are:

- `Engine`—Either `redis` or `memcached`
- `CacheNodeType`—Similar to the EC2 instance type, for example `cache.t2.micro`
- `NumCacheNodes`—1 for a single-node cluster
- `CacheSubnetGroupName`—You reference subnets of a VPC using a dedicated resource called a subnet group
- `VpcSecurityGroupIds`—The security groups you want to attach to the cluster

A minimal CloudFormation template is shown in listing 12.1.

Listing 12.1 Minimal CloudFormation template of an ElastiCache Redis single-node cluster

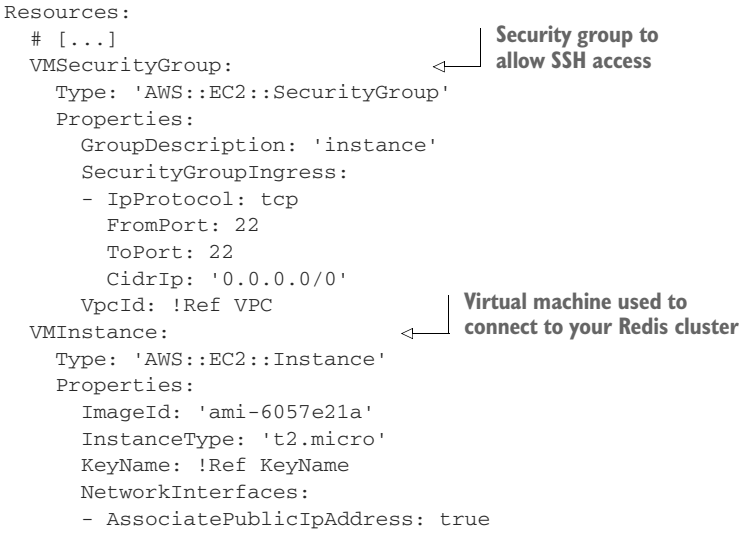
```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 12 (minimal)'
Parameters:                                    <-- Defines VPC and subnets as parameters
  VPC:
    Type: 'AWS::EC2::VPC::Id'
  SubnetA:
    Type: 'AWS::EC2::Subnet::Id'
  SubnetB:
    Type: 'AWS::EC2::Subnet::Id'
  KeyName:
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
Resources:
  CacheSecurityGroup:                          <-- The security group to manage which traffic is allowed to enter/leave the cluster
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
```



As already mentioned, ElastiCache nodes in a cluster only have private IP addresses. Therefore, you can't connect to a node directly over the internet. The same is true for other resources as EC2 instances or RDS instances. To test the Redis cluster, you can create an EC2 instance in the same VPC as the cluster. From the EC2 instance, you can then connect to the private IP address of the cluster.

12.1.2 Test the Redis cluster

To test Redis, add the following resources to the minimal CloudFormation template:



```
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref VMSecurityGroup
        SubnetId: !Ref SubnetA
Outputs:
  VMInstanceIPAddress:
    Value: !GetAtt 'VMInstance.PublicIp'
    Description: 'EC2 Instance public IP address
➥ (connect via SSH as user ec2-user)'
  CacheAddress:
    Value: !GetAtt 'Cache.RedisEndpoint.Address'
    Description: 'Redis DNS name (resolves to a private IP address)'
```

Public IP address of virtual machine

DNS name of Redis cluster node (resolves to a private IP address)

The minimal CloudFormation template is now complete. Create a stack based on the template to create all the resources in your AWS account using the Management Console: http://mng.bz/Cp44. You have to fill in four parameters when creating the stack:

- `KeyName`—If you've been following along with our book in order, you should have created a key pair with the name `mykey`, which is selected by default.
- `SubnetA`—You should have at least two options here; select the first one.
- `SubnetB`—You should have at least two options here; select the second one.
- `VPC`—You should only have one possible VPC here—your default VPC. Select it.

You can find the full code for the template at /chapter12/minimal.yaml in the book's code folder.

> **Where is the template located?**
> You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at chapter12/minimal.yaml. On S3, the same file is located at http://mng.bz/qJ8g.

Once the stack status changes to CREATE_COMPLETE in the CloudWatch Management Console, select the stack and click on the Outputs tab. You can now start to test the Redis cluster. Open an SSH connection to the EC2 instance, and then you can use the Redis CLI to interact with the Redis cluster node.

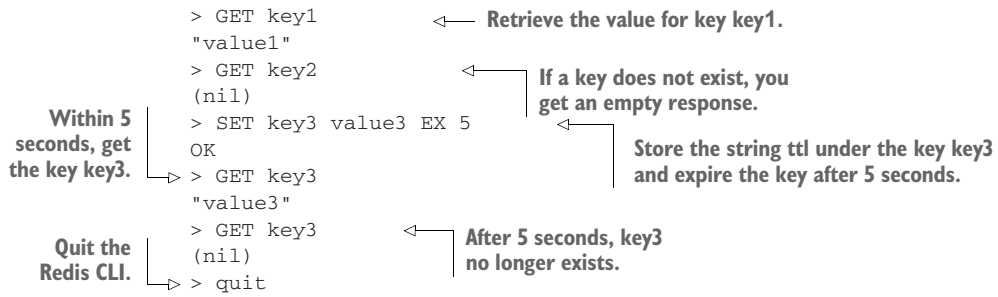
Connect to the EC2 instance, replace $VMInstanceIPAddress with the output from the CloudFormation stack.

Install the Redis CLI.

Store the string value under the key key1.

```
$ ssh -i mykey.pem ec2-user@$VMInstanceIPAddress
$ sudo yum -y install --enablerepo=epel redis
$ redis-cli -h $CacheAddress
> SET key1 value1
OK
```

Connect to the Redis cluster node, replace $CacheAddress with the output from the CloudFormation stack.



You've successfully connected to a Redis cluster node, stored some keys, retrieved some keys, and used Redis's time-to-live functionality. With this knowledge, you could start to implement a caching layer in your own application. But as always, there are more options to discover. Delete the CloudFormation stack you created to avoid unwanted costs. Then, continue with the next section to learn more about advanced deployment options with more than one node to achieve high availability or sharding.

12.2 Cache deployment options

Which deployment option you should choose is influenced by four factors:

1. *Engine*—Memcached or Redis
2. *Backup/Restore*—Is it possible to back up or restore the data from the cache?
3. *Replication*—If a single node fails, is the data still available?
4. *Sharding*—If the data does not fit on a single node, can you add nodes to increase capacity?

Table 12.2 compares the deployment options for the two available engines.

Table 12.2 Comparing ElastiCache deployment options

	Memcached	Redis: single node	Redis: cluster mode disabled	Redis: cluster mode enabled
Backup/Restore	no	yes	yes	yes
Replication	no	no	yes	yes
Sharding	yes	no	no	yes

Let's look at deployment options in more detail.

12.2.1 Memcached: cluster

An Amazon ElastiCache for a Memcached cluster consists of 1-20 nodes. Sharding is implemented by the Memcached client, typically utilizing a consistent hashing algorithm which arranges keys into partitions in a ring distributed across the nodes. The client essentially decides which keys belong to which nodes and directs the requests to those partitions. Each node stores a unique portion of the key-space in-memory. If a

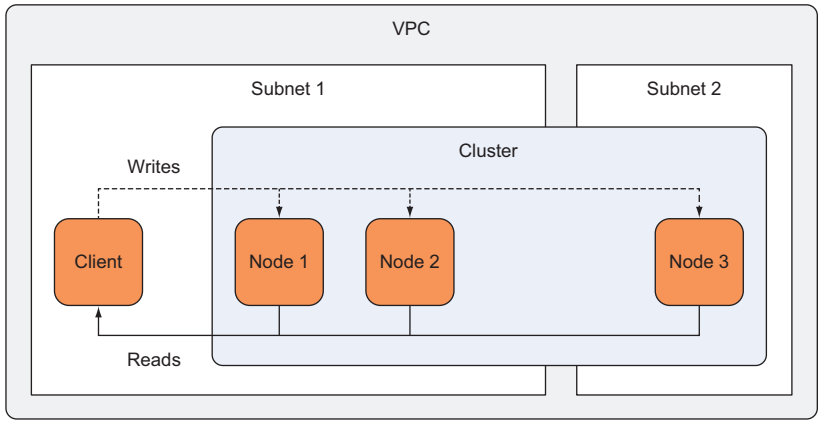


Figure 12.4 Memcached deployment option: cluster

node fails, the node is replaced but the data is lost. You can not back up the data in Memcached. Figure 12.4 shows a Memcached cluster deployment.

You can use a Memcached cluster if your application requires a simple in-memory store and can tolerate the loss of a node and its data. The SQL cache example in the beginning of this chapter could be implemented using Memcached. Since the data is always available in the relational database, you can tolerate a node loss, and you only need simple commands (GET, SET) to implement the query cache.

12.2.2 Redis: Single-node cluster

An ElastiCache for a Redis single-node cluster always consists of one node. Sharding and high availability are not possible with a single node. But Redis supports the creation of backups, and also allows you to restore those backups. Figure 12.5 shows a Redis single-node cluster. Remember that a VPC is a way to define a private network on AWS. A subnet is a way to separate concerns inside the VPC. Cluster nodes always run in a single subnet. The client communicates with the Redis cluster node to get data and write data to the cache.

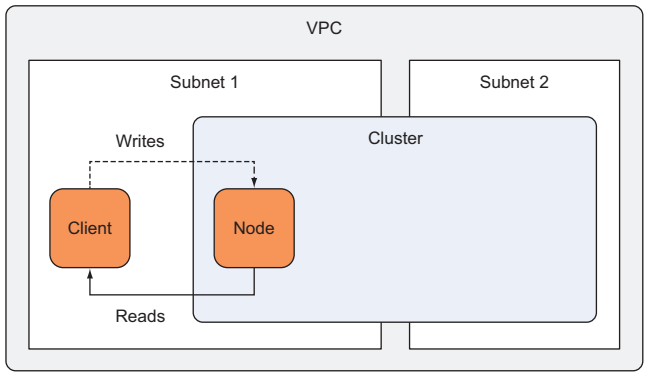


Figure 12.5 Redis deployment option: single-node cluster

A single node adds a single point of failure (SPOF) to your system. This is probably something you want to avoid for business-critical production systems.

12.2.3 Redis: Cluster with cluster mode disabled

Things become more complicated now, because ElastiCache uses two terminologies. We've been using the terms cluster/node/shard so far, and the graphical Management Console also uses these terms. But the API, the CLI, and CloudFormation use a different terminology: replication group/node/node group. We prefer the cluster/node/shard terminology, but in figures 12.6 and 12.7 we've added the replication group/node/node group terminology in parentheses.

A Redis cluster with cluster mode disabled supports backups and data replication, but no sharding. This means there is only one shard consisting of one primary and up to five replica nodes.

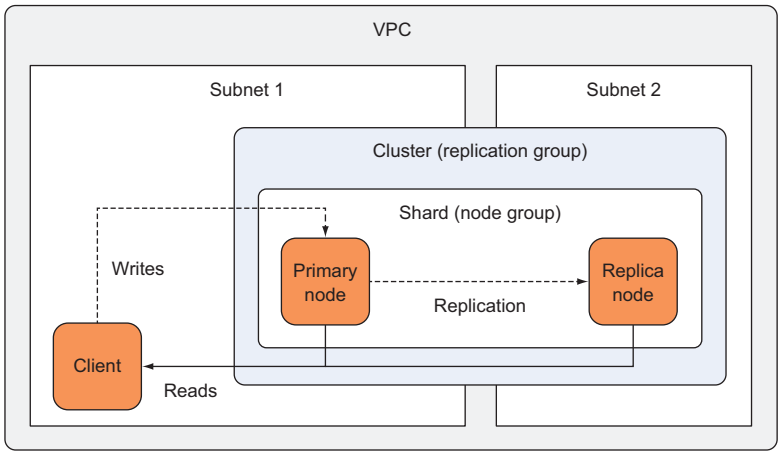


Figure 12.6 Redis deployment option: cluster with cluster mode disabled

You can use a Redis cluster with cluster mode disabled when you need data replication and all your cached data fits into the memory of a single node. Imagine that your cached data set is 4 GB in size. If your cache has at least 4 GB of memory, the data fits into the cache and you don't need sharding.

12.2.4 Redis: Cluster with cluster mode enabled

A Redis cluster with cluster mode enabled supports backups, data replication, and sharding. You can have up to 15 shards per cluster. Each shard consists of one primary and up to five replica nodes. The largest cluster size therefore is 90 nodes (15 primaries + (15 * 5 replicas)).

You can use a Redis cluster with cluster mode enabled when you need data replication and your data is too large to fit into the memory of a single node. Imagine that your cached data is 22 GB in size. Each cache node has a capacity of 4 GB of memory.

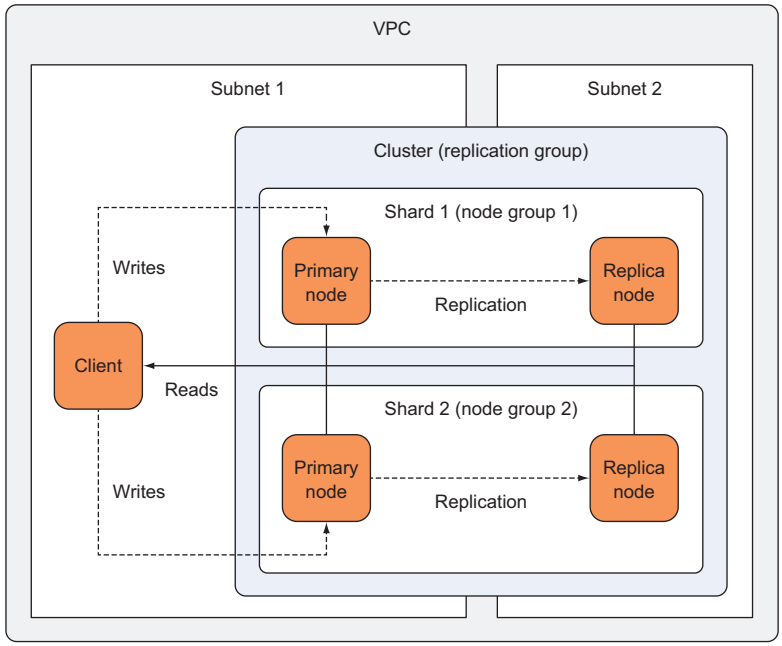


Figure 12.7 Redis deployment option: cluster with cluster mode enabled

Therefore, you will need six shards to get a total capacity of 24 GB of memory. ElastiCache provides up to 437 GB of memory per node, which totals to a maximum cluster capacity of 6.5 TB (15 * 437 GB).

Additional benefits of enabling cluster mode

With cluster mode enabled, failover speed is much faster, as no DNS is involved. Clients are provided a single configuration endpoint to discover changes to the cluster topology, including newly elected primaries. With cluster mode disabled, AWS provides a single primary endpoint and in the event of a failover, AWS does a DNS swap on that endpoint to one of the available replicas. It may take ~1–1.5min before the application is able to reach the cluster after a failure, whereas with cluster mode enabled, the election takes less than 30s.

More shards enable more read/write performance. If you start with one shard and add a second shard, each shard now only has to deal with 50% of the requests (assuming an even distribution).

As you add nodes, your blast radius decreases. For example, if you have five shards and experience a failover, only 20% of your data is affected. This means you can't write to this portion of the key space until the failover process completes (~15–30s), but you can still read from the cluster, given you have a replica available. With cluster mode disabled, 100% of your data is affected, as a single node consists of your entire key space. You can read from the cluster but can't write until the DNS swap has completed.

You are now equipped to select the right engine and the deployment option for your use case. In the next section, you will take a closer look at the security aspects of ElastiCache to control access to your cache cluster.

12.3 Controlling cache access

Access control is very similar to the way it works with RDS (see section 11.4). The only difference is, that cache engines come with very limited features to control access to the data itself. The following paragraph summarizes the most important aspects of access control.

ElastiCache is protected by four layers:

- *Identity and Access Management (IAM):* Controls which IAM user/group/role is allowed administer an ElastiCache cluster.
- *Security Groups:* Restricts incoming and outgoing traffic to ElastiCache nodes.
- *Cache Engine:* Redis has the AUTH command, Memcached does not handle authentication. Neither engine supports authorization.
- *Encryption:* At rest and in transit.

SECURITY WARNING It's important to understand that you don't control access to the cache nodes using IAM. Once the nodes are created, Security Groups control the access.

12.3.1 Controlling access to the configuration

Access to the ElastiCache service is controlled with the help of the IAM service. The IAM service is responsible for controlling access to actions like creating, updating, and deleting a cache cluster. IAM doesn't manage access inside the cache; that's the job of the cache engine. An IAM policy defines the configuration and management actions a user, group, or role is allowed to execute on the ElastiCache service. Attaching the IAM policy to IAM users, groups, or roles controls which entity can use the policy to configure an ElastiCache cluster.

You can get a complete list of IAM actions and resource-level permissions supported at http://mng.bz/anNF.

12.3.2 Controlling network access

Network access is controlled with security groups. Remember the security group from the minimal CloudFormation template in section 12.1 where access to port 6379 (Redis) was allowed for all IP addresses. But since cluster nodes only have private IP addresses this restricts access to the VPC:

```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
```

```
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        CidrIp: '0.0.0.0/0'
```

You should improve this setup by working with two security groups. To control traffic as tight as possible, you will not white list IP addresses. Instead, you create two security groups. The client security group will be attached to all EC2 instances communicating with the cache cluster (your web servers). The cache cluster security group allows inbound traffic on port 6379 only for traffic that comes from the client security group. This way you can have a dynamic fleet of clients who is allowed to send traffic to the cache cluster. You used the same approach for the SSH bastion host in section 6.4.

```
Resources:
  # [...]
  ClientSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'cache-client'
      VpcId: !Ref VPC
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
      SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 6379
        ToPort: 6379
        SourceSecurityGroupId: !Ref ClientSecurityGroup
```

Only allow access from the ClientSecurityGroup.

Attach the `ClientSecurityGroup` to all EC2 instances that need access to the cache cluster. This way, you only allow access to the EC2 instances that really need access.

Keep in mind that ElastiCache nodes always have private IP addresses. This means that you can't accidentally expose a Redis or Memcached cluster to the internet. You still want to use Security Groups to implement the principle of least privilege.

12.3.3 Controlling cluster and data access

Both Redis and Memcached support very basic authentication features. Amazon ElastiCache does support Redis AUTH for customers who also want to enable token-based authentication in additional to the security features Amazon ElastiCache provides. Redis AUTH is the security mechanism that open source Redis utilizes. Since the communication between the clients and the cluster is unencrypted, such an authentication would not improve security using open source engines. But, Amazon ElastiCache does offer encryption in transit with Redis 3.2.6.

Neither engine implements data access management. When connected to a key-value store, you get access to all data. This limitation relies on the underlying key-value stores, not in ElastiCache itself. In the next section, you'll learn how to use ElastiCache for Redis in a real-world application called Discourse.

12.4 Installing the sample application Discourse with CloudFormation

Small communities, like football clubs, reading circles, or dog schools benefit from having a place where members can communicate with each other. Discourse is open-source software for providing modern forums for your community. You can use it as a mailing list, discussion forum, long-form chat room, and more. It is written in Ruby using the Rails framework. Figure 12.8 gives you an impression of Discourse. Wouldn't that be a perfect place for your community to meet? In this section, you will learn how to set up Discourse with CloudFormation. Discourse is also perfectly suited for learning about ElastiCache because it requires a Redis cache. Discourse requires PostgreSQL as main data store and uses Redis to cache data and process transient data.

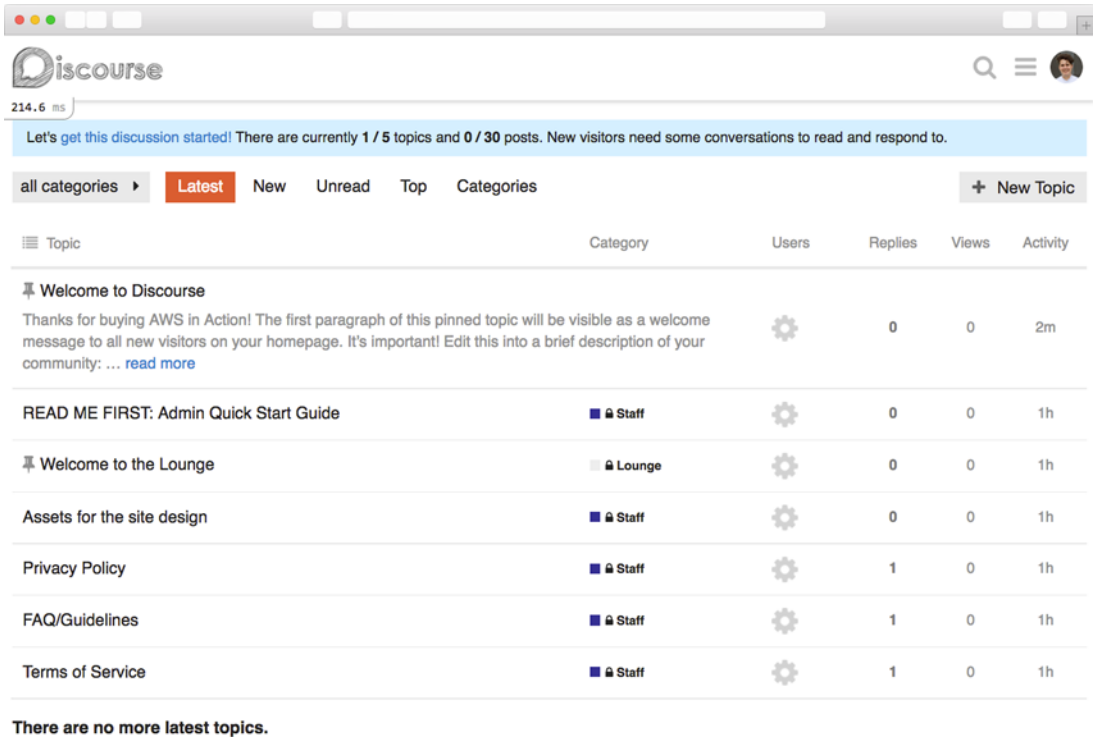


Figure 12.8 Discourse: a platform for community discussion

In this section, you'll create a CloudFormation template with all the components necessary to run Discourse. Finally, you'll create a CloudFormation stack based on the template to test your work. The necessary components are:

- *VPC*—Network configuration
- *Cache*—Security group, subnet group, cache cluster
- *Database*—Security group, subnet group, database instance
- *Virtual machine*—Security group, EC2 instance

Let's get started. You'll start with the first component and extend the template in the rest of this section.

12.4.1 VPC: Network configuration

In section 6.5 you learned all about private networks on AWS. If you can't follow listing 12.2, you could go back to section 6.5 or continue with the next step—understanding the network is not key to get Discourse running.

Listing 12.2 CloudFormation template for Discourse: VPC

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 12'
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
  AdminEmailAddress:
    Description: 'Email address of admin user'
    Type: 'String'
Resources:
  VPC:
    Type: 'AWS::EC2::VPC'
    Properties:
      CidrBlock: '172.31.0.0/16'
      EnableDnsHostnames: true
  InternetGateway:
    Type: 'AWS::EC2::InternetGateway'
    Properties: {}
  VPCGatewayAttachment:
    Type: 'AWS::EC2::VPCGatewayAttachment'
    Properties:
      VpcId: !Ref VPC
      InternetGatewayId: !Ref InternetGateway
  SubnetA:
    Type: 'AWS::EC2::Subnet'
    Properties:
      AvailabilityZone: !Select [0, !GetAZs '']
      CidrBlock: '172.31.38.0/24'
      VpcId: !Ref VPC
  SubnetB: # [...]
```

The key pair name for SSH access and also the email address of the Discourse admin (must be valid!) are variable.

Creates a VPC in the address range 172.31.0.0/16

We want to access Discourse from the internet, so we need an internet gateway.

Attach the internet gateway to the VPC.

Create a subnet in the address range 172.31.38.0/24 in the first availability zone (array index 0).

Create a second subnet in the address range 172.31.37.0/24 in the second availability zone (properties omitted).

```
RouteTable:
  Type: 'AWS::EC2::RouteTable'
  Properties:
    VpcId: !Ref VPC
SubnetRouteTableAssociationA:
  Type: 'AWS::EC2::SubnetRouteTableAssociation'
  Properties:
    SubnetId: !Ref SubnetA
    RouteTableId: !Ref RouteTable
RouteToInternet:
  Type: 'AWS::EC2::Route'
  Properties:
    RouteTableId: !Ref RouteTable
    DestinationCidrBlock: '0.0.0.0/0'
    GatewayId: !Ref InternetGateway
  DependsOn: VPCGatewayAttachment
SubnetRouteTableAssociationB: # [...]
NetworkAcl:
  Type: AWS::EC2::NetworkAcl
  Properties:
    VpcId: !Ref VPC
SubnetNetworkAclAssociationA:
  Type: 'AWS::EC2::SubnetNetworkAclAssociation'
  Properties:
    SubnetId: !Ref SubnetA
    NetworkAclId: !Ref NetworkAcl
SubnetNetworkAclAssociationB: # [...]
NetworkAclEntryIngress:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAcl
    RuleNumber: 100
    Protocol: -1
    RuleAction: allow
    Egress: false
    CidrBlock: '0.0.0.0/0'
NetworkAclEntryEgress:
  Type: 'AWS::EC2::NetworkAclEntry'
  Properties:
    NetworkAclId: !Ref NetworkAcl
    RuleNumber: 100
    Protocol: -1
    RuleAction: allow
    Egress: true
    CidrBlock: '0.0.0.0/0'
```

Create a route table that contains the default route, which routes all subnets in a VPC.

Associate the first subnet with the route table.

Add a route to the internet via the internet gateway.

Create an empty network ACL.

Associate the first subnet with the network ACL.

Allow all incoming traffic on the Network ACL (you will use security groups later as a firewall).

Allow all outgoing traffic on the Network ACL.

The network is now properly configured using two public subnets. Let's configure the cache next.

12.4.2 Cache: Security group, subnet group, cache cluster

You will add the ElastiCache for Redis cluster now. You learned how to describe a minimal cache cluster earlier in this chapter. This time, you'll add a few extra properties

to enhance the setup. This listing contains the CloudFormation resources related to the cache.

Listing 12.3 CloudFormation template for Discourse: Cache

```
Resources:
  # [...]
  CacheSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: cache
      VpcId: !Ref VPC
  CacheSecurityGroupIngress:
    Type: 'AWS::EC2::SecurityGroupIngress'
    Properties:
      GroupId: !Ref CacheSecurityGroup
      IpProtocol: tcp
      FromPort: 6379
      ToPort: 6379
      SourceSecurityGroupId: !Ref VMSecurityGroup
  CacheSubnetGroup:
    Type: 'AWS::ElastiCache::SubnetGroup'
    Properties:
      Description: cache
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
  Cache:
    Type: 'AWS::ElastiCache::CacheCluster'
    Properties:
      CacheNodeType: 'cache.t2.micro'
      CacheSubnetGroupName: !Ref CacheSubnetGroup
      Engine: redis
      EngineVersion: '3.2.4'
      NumCacheNodes: 1
      VpcSecurityGroupIds:
      - !Ref CacheSecurityGroup
```

The security group to control incoming and outgoing traffic to/from the cache

To avoid a cyclic dependency, the ingress rule is split into a separate CloudFormation resource.

Redis runs on port 6379.

The VMSecurityGroup resource is not yet specified; you will ad this later when you define the EC2 instance that runs the web server.

The cache subnet group references the VPC subnets.

Create a single-node Redis cluster.

You can specify the exact version of Redis that you want to run. Otherwise the latest version is used, which may cause incompatibility issues in the future. We recommend always specifying the version.

The single-node Redis cache cluster is now defined. Discourse also requires a PostgreSQL database, which you'll define next.

12.4.3 Database: Security group, subnet group, database instance

PostgreSQL is a powerful, open source, and relational database. If you are not familiar with PostgreSQL, that's not a problem at all. Luckily, the RDS service will provide a managed PostgreSQL database for you. You learned about RDS in chapter 11. Listing 12.4 shows the section of the template that defines the RDS instance.

Listing 12.4 CloudFormation template for Discourse: Database

```
Resources:
  # [...]
  DatabaseSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: database
      VpcId: !Ref VPC
  DatabaseSecurityGroupIngress:
    Type: 'AWS::EC2::SecurityGroupIngress'
    Properties:
      GroupId: !Ref DatabaseSecurityGroup
      IpProtocol: tcp
      FromPort: 5432
      ToPort: 5432
      SourceSecurityGroupId: !Ref VMSecurityGroup
  DatabaseSubnetGroup:
    Type: 'AWS::RDS::DBSubnetGroup'
    Properties:
      DBSubnetGroupDescription: database
      SubnetIds:
      - Ref: SubnetA
      - Ref: SubnetB
  Database:
    Type: 'AWS::RDS::DBInstance'
    Properties:
      AllocatedStorage: 5
      BackupRetentionPeriod: 0
      DBInstanceClass: 'db.t2.micro'
      DBName: discourse
      Engine: postgres
      EngineVersion: '9.5.6'
      MasterUsername: discourse
      MasterUserPassword: discourse
      VPCSecurityGroups:
      - !Sub ${DatabaseSecurityGroup.GroupId}
      DBSubnetGroupName: !Ref DatabaseSubnetGroup
    DependsOn: VPCGatewayAttachment
```

Traffic to/from the RDS instance is protected by a security group.

PostgreSQL runs on port 5432 by default.

The VMSecurityGroup resource is not yet specified; you'll add this later when you define the EC2 instance that runs the web server.

RDS also uses a subnet group to reference the VPC subnets.

The database resource

Disable backups; you want to turn this on (value > 0) in production.

RDS created a database for you in PostgreSQL.

Discourse requires PostgreSQL.

We recommend to always specify the version of the engine to avoid future incompatibility issues.

PostgreSQL admin user name

PostgreSQL admin password; you want to change this in production.

Have you noticed the similarity between RDS and ElastiCache? The concepts are similar, which makes it easier for you to work with both services. Only one component is missing: the EC2 instance that runs the web server.

12.4.4 Virtual machine—security group, EC2 instance

Discourse is a Ruby on Rails application so you need an EC2 instance to host the application. Listing 12.5 defines the virtual machine and the startup script to install and configure Discourse.

Listing 12.5 CloudFormation template for Discourse: Virtual machine

```
Resources:
  # [...]
  VMSecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      GroupDescription: 'vm'
      SecurityGroupIngress:
      - CidrIp: '0.0.0.0/0'
        FromPort: 22
        IpProtocol: tcp
        ToPort: 22
      - CidrIp: '0.0.0.0/0'
        FromPort: 80
        IpProtocol: tcp
        ToPort: 80
      VpcId: !Ref VPC
  VMInstance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: 't2.micro'
      KeyName: !Ref KeyName
      NetworkInterfaces:
      - AssociatePublicIpAddress: true
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - !Ref VMSecurityGroup
        SubnetId: !Ref SubnetA
      UserData:
        'Fn::Base64': !Sub |
          #!/bin/bash -x
          bash -ex << "TRY"
            # [...]

            # download Discourse
            useradd discourse
            mkdir /opt/discourse
            git clone https://github.com/AWSinAction/discourse.git \
➥ /opt/discourse

            # configure Discourse
            echo "db_host = \"${Database.Endpoint.Address}\"" >> \
➥ /opt/discourse/config/discourse.conf
            echo "redis_host = \"${Cache.RedisEndpoint.Address}\"" >> \
➥ /opt/discourse/config/discourse.conf
            # [...]
          TRY
          /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
➥ --resource VMInstance --region ${AWS::Region}
      CreationPolicy:
        ResourceSignal:
```

Allow SSH traffic from the public internet.

Allow HTTP traffic from the public internet.

The virtual machine that runs Discourse

Only contains an excerpt of the full script necessary to install Discourse. You can find the full code at /chapter12/template.yaml in the book's code folder.

Configure the Redis cluster node endpoint.

Download Discourse.

Configure the PostgreSQL database endpoint.

Signal the end of the installation script back to CloudFormation.

```
        Timeout: PT15M
    DependsOn:
    - VPCGatewayAttachment
Outputs:
  VMInstanceIPAddress:
    Value: !GetAtt 'VMInstance.PublicIp'
    Description: 'EC2 Instance public IP address
➥ (connect via SSH as user ec2-user)'
```

Wait up to 15 minutes for the signal from the install script in UserData.

Output the public IP address of the virtual machine.

You've reached the end of the template. All components are defined now. It's time to create a CloudFormation stack based on your template to see if it works.

12.4.5 Testing the CloudFormation template for Discourse

Let's create a stack based on your template to create all the resources in your AWS account. To find the full code for the template, go to /chapter12/template.yaml in the book's code folder. Use the AWS CLI to create the stack:

```
$ aws cloudformation create-stack --stack-name discourse \
➥ --template-url https://s3.amazonaws.com/awsinaction-code2/\
➥ chapter12/template.yaml \
➥ --parameters ParameterKey=KeyName,ParameterValue=mykey \
➥ "ParameterKey=AdminEmailAddress,ParameterValue=your@mail.com"
```

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at chapter12/template.yaml. On S3, the same file is located at http://mng.bz/jP32.

The creation of the stack can take up to 15 minutes. To check the status of the stack, use the following command:

```
$ aws cloudformation describe-stacks --stack-name discourse \
➥ --query "Stacks[0].StackStatus"
```

If the stack status is CREATE_COMPLETE, the next step is to get the public IP address of the EC2 instance from the stack's outputs with the following command:

```
$ aws cloudformation describe-stacks --stack-name discourse \
➥ --query "Stacks[0].Outputs[0].OutputValue"
```

Open a web browser and insert the IP address in the address bar to open your Discourse website. Figure 12.9 shows the website. Click Register to create an admin account.

You will receive an email to activate your account. This email will likely be in your spam folder! After activation, the 13-step setup wizard is started, which you have to

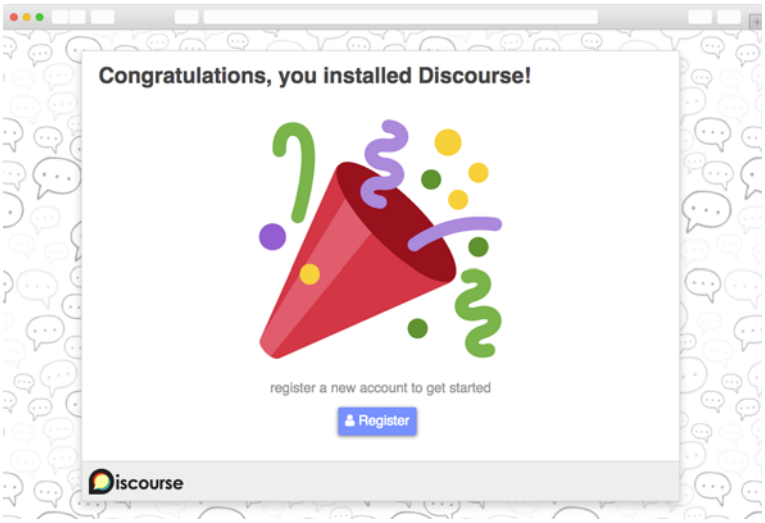


Figure 12.9 Discourse: first screen after a fresh install

complete. After you complete the wizard and have successfully installed Discourse, the screen shown in figure 12.10 should appear.

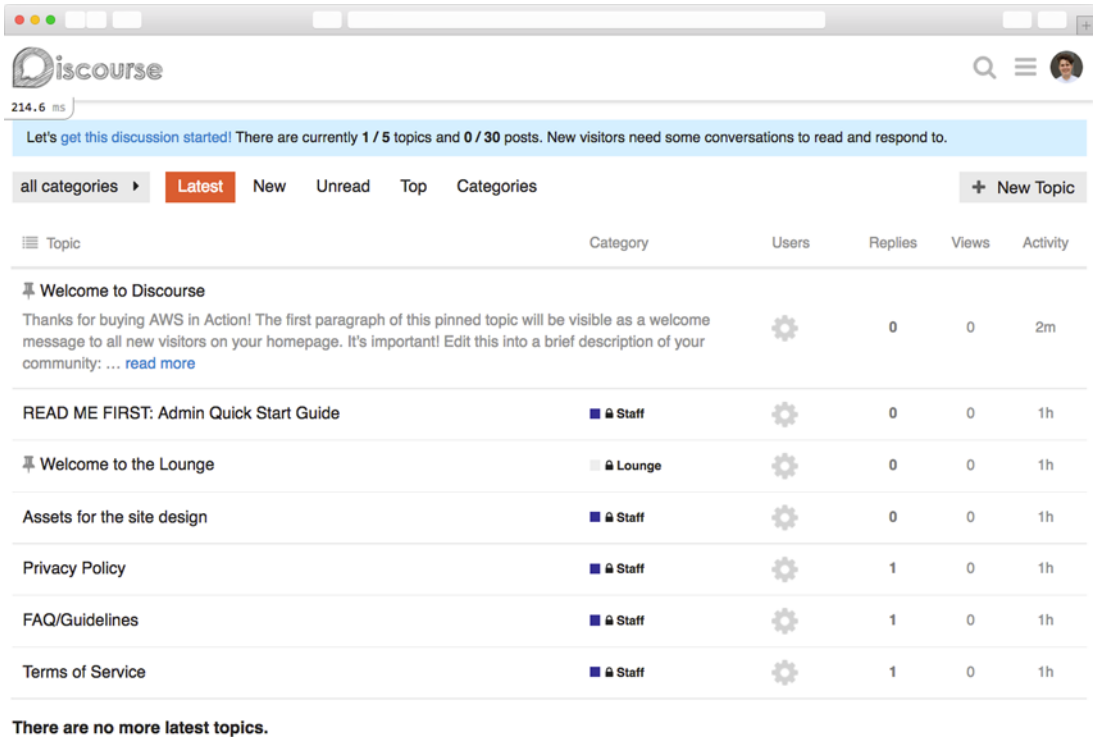


Figure 12.10 Discourse: a platform for community discussion

Don't delete the CloudFormation stack because you'll use the setup in the next section.

You learned how Discourse uses Redis to cache some data while using ElastiCache to run Redis for you. If you use ElastiCache in production, you also have to set up monitoring to ensure that the cache works as expected, and you have to know how to improve performance. Those are the topics of the next two chapters.

12.5 Monitoring a cache

CloudWatch is the service on AWS that stores all kinds of metrics. ElastiCache nodes send useful metrics. The most important metrics to watch are:

- `CPUUtilization`—The percentage of CPU utilization.
- `SwapUsage`—The amount of swap used on the host, in bytes. *Swap* is space on disk that is used if the system runs out of physical memory.
- `Evictions`—The number of non-expired items the cache evicted due to the memory limit.
- `ReplicationLag`—This metric is only applicable for a Redis node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. Usually this number is very low.

In this section we'll examine those metrics in more detail, and give you some hints about useful thresholds for defining alarms on those metrics to set up production-ready monitoring for your cache.

12.5.1 Monitoring host-level metrics

The virtual machines report CPU utilization and swap usage. CPU utilization usually gets problematic when crossing 80–90%, because the wait time explodes. But things are more tricky here. Redis is single-threaded. If you have many cores, the overall CPU utilization can be low but one core can be at 100% utilization. Swap usage is a different topic. You run an in-memory cache, so if the virtual machine starts to swap (move memory to disk) the performance will suffer. By default, ElastiCache for Memcached and Redis is configured to limit memory consumption to a value smaller than what's physical available (you can tune this) to have room for other resources (for example, the kernel needs memory for each open socket). But other processes (such as kernel processes) are also running, and they may start to consume more memory than what's available. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards.

> **Queuing theory: why 80–90%?**
>
> Imagine you are the manager of a supermarket. What should be the goal for daily utilization of your cashier? It's tempting to go for a high number. Maybe 90%. But it turns out that the wait time for your customers is very high when your cashiers are utilized for 90% of the day, because customers don't arrive at the same time at the queue.

The theory behind this is called *queuing theory*, and it turns out that wait time is exponential to the utilization of a resource. This not only applies to cashiers, but also to network cards, CPU, hard disks, and so on. Keep in mind that this sidebar simplifies the theory and assumes an M/D/1 queuing system: Markovian arrivals (exponentially distributed arrival times), deterministic service times (fixed), one service center. To you want to learn more about queuing theory applied to computer systems, we recommend *Systems Performance: Enterprise and the Cloud* by Brendan Gregg (Prentice Hall, 2013) to get started.

When you go from 0% utilization to 60%, wait time doubles. When you go to 80%, wait time has tripled. When you to 90%, wait time is six times higher. And so on.

So if your wait time is 100 ms during 0% utilization, you already have 300 ms wait time during 80% utilization, which is already slow for a e-commerce web site.

You might set up an alarm to trigger if the 10-minute average of the `CPUUtilization` metric is higher than 80% for 1 out of 1 data points, and if the 10-minute average of the `SwapUsage` metric is higher than 67108864 (64 MB) for 1 out of 1 datapoints. These numbers are just a rule of thumb. You should load-test your system to verify that the thresholds are high/low enough to trigger the alarm before application performance suffers.

12.5.2 Is my memory sufficient?

The `Evictions` metric is reported by Memcached and Redis. If the cache is full and you put a new key-value pair into the cache, an old key-value pair needs to be deleted first. This is called an eviction. Redis only evicts keys with a TTL by default (volatile-lru). These additional eviction strategies are available: allkeys-lru (remove the least recently used key among all keys), volatile-random (remove a random key among keys with TTL), allkeys-random (remove a random key among all keys), volatile-ttl (remove the key with the shortest TTL), and noeviction (do not evict any key). Usually, high eviction rates are a sign that you either aren't using a TTL to expire keys after some time, or that your cache is too small. You can solve this issue by increasing the memory of the cache, either by increasing the node type or by adding more shards.

You might set an alarm to trigger if the 10-minute average of the `Evictions` metric is higher than 1000 for 1 out of 1 data points.

12.5.3 Is my Redis replication up-to-date?

The `ReplicationLag` metric is only applicable for a node running as a read replica. It represents how far behind, in seconds, the replica is in applying changes from the primary node. The higher this value, the more out-of-date the replica is. This can be a problem because some users of your application will see very old data. In the gaming application, imagine you have one primary node and one replica node. All reads are performed by either the primary or the replica node. The `ReplicationLag` is 600,

which means that the replication node looks like the primary node looked 10 minutes before. Depending on which node the user hits when accessing the application, they could see 10-minute old data.

What are reasons for a high ReplicationLag? There could be a problem with the sizing of your cluster; for example, your cache cluster might be at capacity. Typically this will be a sign to increase the capacity by adding shards or replicas.

You might set an alarm to trigger if the 10-minute average of the `ReplicationLag` metric is higher than 30 for 1 consecutive period.

Cleaning up

It's time to delete the running CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name discourse
```

12.6 Tweaking cache performance

Your cache can become a bottleneck if it can no longer handle the requests with low latency. In the previous section, you learned how to monitor your cache. In this section you learn what you can do if your monitoring data shows that your cache is becoming the bottleneck (for example if you see high CPU or network usage). Figure 12.11 contains a decision tree that you can use to resolve performance issues with ElastiCache. The strategies are described in more detail in the rest of this section.

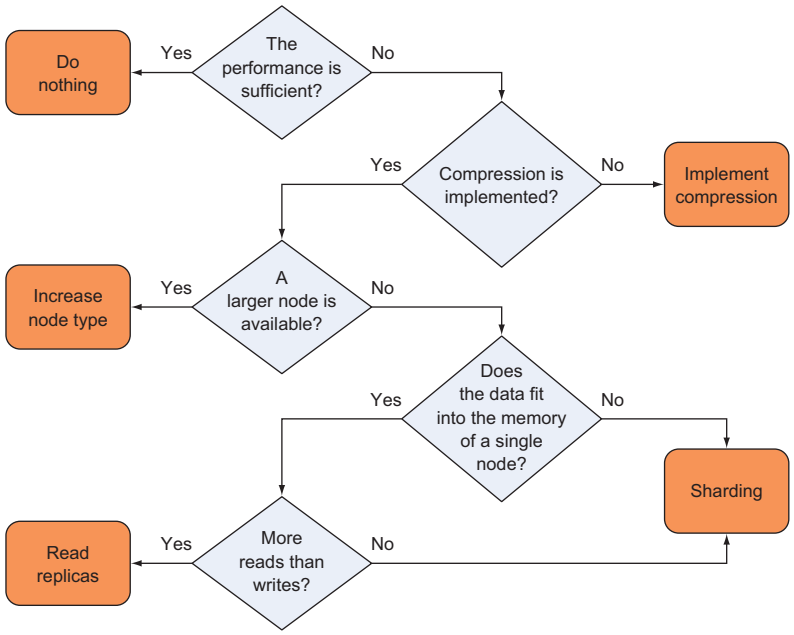


Figure 12.11 ElastiCache decision tree to resolve performance issues

There are three strategies for tweaking the performance of your ElastiCache cluster:

1. *Selecting the right cache node type*—A bigger instance type comes with more resources (CPU, memory, network) so you can scale vertically.
2. *Selecting the right deployment option*—You can use sharding or read replicas to scale horizontally.
3. *Compressing your data*—If you shrink the amount of data being transferred and stored, you can also tweak performance.

12.6.1 Selecting the right cache node type

So far, you used the cache node type `cache.t2.micro`, which comes with one vCPU, ~0.6 GB memory, and low-to-moderate network performance. You used this node type because it's part of the Free Tier. But you can also use more powerful node types on AWS. The upper end is the `cache.r4.16xlarge` with 64 vCPUs, ~488 GB memory, and 25 Gb network. Keep in mind that Redis is single-threaded and will not use all cores.

As a rule of thumb: for production traffic, select a cache node type with at least 2 vCPUs for real concurrency, enough memory to hold your data set with some space to grow (say, 20%; this also avoids memory fragmentation), and at least high network performance. The `r4.large` is an excellent choice for a small node size: 2 vCPUs, ~16 GB, and up to 10 Gb of network. This may be a good starting point when considering how many shards you may want in a clustered topology, and if you need more memory, move up a node type. You can find the available node types at https:// aws.amazon.com/elasticache/pricing/.

12.6.2 Selecting the right deployment option

By replicating data, you can distribute read traffic to multiple nodes within the same replica group. Since you have more nodes in your cluster, you can serve more requests. By sharding data, you split the data into multiple buckets. Each bucket contains a subset of the data. Since you have more nodes in your cluster, you can serve more requests.

You can also combine replication and sharding to increase the number of nodes in your cluster.

Both Memcached and Redis support the concept of sharding. With sharding, a single cache cluster node is no longer responsible for all the keys. Instead the key space is divided across multiple nodes. Both Redis and Memcached clients implement a hashing algorithm to select the right node for a given key. By sharding, you can increase the capacity of your cache cluster.

Redis supports the concept of replication, where one node in a node group is the primary node accepting read and write traffic, while the replica nodes only accept read traffic. This allows you the scale the read capacity. The Redis client has to be aware of the cluster topology to select the right node for a given command. Keep in mind that the replicas are synchronized asynchronously. This means that the replication node eventually reaches the state of the primary node.

As a rule of thumb: when a single node can no longer handle the amount of data or the requests, and if you are using Redis with mostly read traffic, then you should use replication. Replication also increases the availability at the same time (at no extra costs).

12.6.3 Compressing your data

This solution needs to be implemented in your application. Instead of sending large values (and also keys) to your cache, you can compress the data before you store it in the cache. When you retrieve data from the cache, you have to uncompress it on the application before you can use the data. Depending on your data, compressing data can have a significant effect. We saw memory reductions to 25% of the original size and network transfer savings of the same size.

As a rule of thumb: Compress your data using a compression algorithm that is best suited for your data, most likely the zlib library. You have to experiment with a subset of your data to select the best compression algorithm that is also supported by your programming language.

Summary

- A caching layer can speed up your application significantly, while also lowering the costs of your primary data store.
- To keep the cache in sync with the database, items usually expire after some time, or a write-through strategy is used.
- When the cache is full, the least frequently used items are usually evicted.
- ElastiCache can run Memcached or Redis clusters for you. Depending on the engine, different features are available. Memcached and Redis are open source, but AWS added engine-level enhancements.

13 Programming for the NoSQL database service: DynamoDB

This chapter covers

- Advantages and disadvantages of the NoSQL service DynamoDB
- Creating tables and storing data
- Adding secondary indexes to optimize data retrieval
- Designing a data model optimized for a key-value database
- Tuning performance

Most applications depend on a database where data is stored. Imagine an application that keeps track of a warehouse's inventory. The more inventory moves through the warehouse, the more requests the application serves, and the more queries the database has to process. Sooner or later, the database becomes too busy and latency increases to a level that limits the warehouse's productivity. At this point, you have to scale the database to help the business. This can be done in two ways:

- *Vertically*—You can add more hardware to your database machine; for example, you can add memory or replace the CPU with a more powerful model.
- *Horizontally*—You can add a second database machine. Both machines then form a database cluster.

Scaling a database vertically is the easier option, but it gets expensive. High-end hardware is more expensive than commodity hardware. Besides that, at some point, you can no longer add faster hardware because nothing faster is available.

Scaling a traditional, relational database horizontally is difficult because transactional guarantees (atomicity, consistency, isolation, and durability, also known as *ACID*) require communication among all nodes of the database during a two-phase commit. A simplified two-phase commit with two nodes works like this:

1. A query is sent to the database cluster that wants to change data (INSERT, UPDATE, DELETE).
2. The database transaction coordinator sends a commit request to the two nodes.
3. Node 1 checks if the query could be executed. The decision is sent back to the coordinator. If the nodes decides yes, it must fulfill this promise. There is no way back.
4. Node 2 checks if the query could be executed. The decision is sent back to the coordinator.
5. The coordinator receives all decisions. If all nodes decide that the query could be executed, the coordinator instructs the nodes to finally commit.
6. Nodes 1 and 2 finally change the data. At this point, the nodes must fulfill the request. This step must not fail.

The problem is that the more nodes you add, the slower your database becomes, because more nodes must coordinate transactions between each other. The way to tackle this has been to use databases that don't adhere to these guarantees. They're called *NoSQL databases*.

There are four types of NoSQL databases—document, graph, columnar, and key-value store—each with its own uses and applications. Amazon provides a NoSQL database service called *DynamoDB*, a key-value store with document support. Unlike RDS, which effectively provides several common RDBMS engines like MySQL, MariaDB, Oracle Database, Microsoft SQL Server, and PostgreSQL, DynamoDB is a fully managed, proprietary, closed source key-value store with document support. *Fully managed* means that you only use the service and AWS operates it for you. DynamoDB is highly available and highly durable. You can scale from one item to billions and from one request per second to tens of thousands of requests per second.

If your data requires a different type of NoSQL database—a graph database like Neo4j, for example—you'll need to spin up an EC2 instance and install the database directly on that. Use the instructions in chapters 3 and 4 to do so.

This chapter looks in detail at how to use DynamoDB: both how to administer it like any other service, and how to program your applications to use it. Administering

DynamoDB is simple. You create tables and secondary indexes, and there is only one option to tweak: its read and write capacity, which directly affects its cost and its performance.

We'll look at the basics of DynamoDB and demonstrate them by walking through a simple to-do application called nodetodo, the Hello World of modern applications. Figure 13.1 shows nodetodo in action.

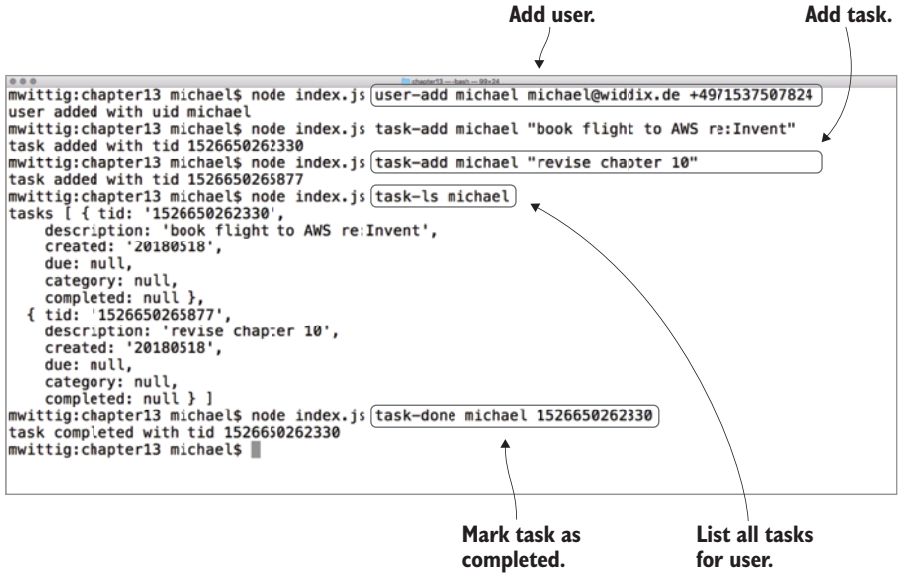


Figure 13.1 You can manage your tasks with the command-line to-do application nodetodo.

> **Examples are 100% covered by the Free Tier**
>
> The examples in this chapter are totally covered by the Free Tier. Keep in mind that this applies only if there is nothing else going on in your AWS account. You'll clean up your account at the end of the chapter.

Before you get started with nodetodo, you need to learn the basics of DynamoDB.

13.1 Operating DynamoDB

DynamoDB doesn't require administration like a traditional relational database, because it's a managed service and AWS takes care of that; instead, you have other tasks to take care of. Pricing depends mostly on your storage use and performance requirements. This section also compares DynamoDB to RDS.

13.1.1 Administration

With DynamoDB, you don't need to worry about installation, updates, machines, storage, or backups. Here's why:

- *DynamoDB isn't software you can download.* Instead, it's a NoSQL database as a service. Therefore you can't install DynamoDB like you would MySQL or MongoDB. This also means you don't have to update your database; the software is maintained by AWS.
- *DynamoDB runs on a fleet of machines operated by AWS.* AWS takes care of the OS and all security-related questions. From a security perspective, it's your job to grant the right permissions through IAM to the users of your DynamoDB tables.
- *DynamoDB replicates your data among multiple machines and across multiple data centers.* There is no need for a backup from a durability point of view—the backup is already built into the database.

> **Backups**
>
> DynamoDB provides very high durability. But what if the database administrator accidentally deletes all the data or a new version of the application corrupts items? In this case, you would need a backup to restore to a working table state from the past. In December 2017, AWS announced a new feature for DynamoDB: on-demand backup and restore.
>
> We strongly recommend using on-demand backups to create snapshots of your DynamoDB tables to be able to restore them later, if needed.

Now you know some administrative tasks that are no longer necessary if you use DynamoDB. But you still have things to consider when using DynamoDB in production: creating tables (see section 13.4), creating secondary indexes (section 13.6), monitoring capacity usage, provisioning read and write capacity (section 13.9), and creating backups of your tables.

13.1.2 Pricing

If you use DynamoDB, you pay the following monthly:

- $ 0.25 USD per used GB of storage (secondary indexes consume storage as well)
- $ 0.47 USD per provisioned write-capacity unit of throughput (throughput is explained in section 13.9)
- $ 0.09 USD per provisioned read-capacity unit of throughput

These prices were valid for the North Virginia (us-east-1) region, at the time of this writing. No additional traffic charges apply if you use AWS resources like EC2 instances to access DynamoDB in the same region.

13.1.3 Networking

DynamoDB does not run in your VPC. It is only accessible via the AWS API. You need internet access to talk to the AWS API. This means you can't access DynamoDB from a private subnet, because a private subnet has no route to the internet via an internet gateway. Instead, a NAT gateway is used (see section 6.5 for more details). Keep in mind that an application using DynamoDB can create a lot of traffic, and your NAT gateway is limited to 10 Gbps of bandwidth. A better approach is to set up a VPC endpoint for DynamoDB and use that to access DynamoDB from private subnets without needing a NAT gateway at all. You can read more about VPC endpoints in the AWS documentation at http://mng.bz/c4v6.

13.1.4 RDS comparison

Table 13.1 compares DynamoDB and RDS. Keep in mind that this is like comparing apples and oranges; the only thing DynamoDB and RDS have in common is that both are called databases. Use RDS (or to be more precise, the relational database engines offered by RDS) if your application requires complex SQL queries. Otherwise, you can consider migrating your application to DynamoDB.

Table 13.1 Differences between DynamoDB and RDS

Task	DynamoDB	RDS
Creating a table	Management Console, SDK, or CLI `aws dynamodb create-table`	SQL `CREATE TABLE` statement
Inserting, updating, or deleting data	SDK	SQL `INSERT, UPDATE, or DELETE` statement, respectively
Querying data	If you query the primary key: SDK. Querying non-key attributes isn't possible, but you can add a secondary index or scan the entire table.	SQL `SELECT` statement
Increasing storage	No action needed: DynamoDB grows with your items.	Provision more storage.
Increasing performance	Horizontal, by increasing capacity. DynamoDB will add more machines under the hood.	Vertical, by increasing instance size and disk throughput; or horizontal, by adding read replicas. There is an upper limit.
Installing the database on your machine	DynamoDB isn't available for download. You can only use it as a service.	Download MySQL, MariaDB, Oracle Database, Microsoft SQL Server, or PostgreSQL, and install it on your machine. Aurora is an exception.
Hiring an expert	Search for special DynamoDB skills.	Search for general SQL skills or special skills, depending on the database engine.

13.1.5 NoSQL comparison

Table 13.2 compares DynamoDB to several NoSQL databases. Keep in mind that all of these databases have pros and cons, and the table shows only a high-level comparison of how they can be used on top of AWS.

Table 13.2 Differences between DynamoDB and some NoSQL databases

Task	DynamoDB Key-value store	MongoDB Document store	Neo4j Graph store	Cassandra Columnar store	Riak KV Key-value store
Run the database on AWS in production.	One click: it's a managed service	Self-maintained cluster of EC2 instances, or as a service from a third party	Self-maintained cluster of EC2 instances, or as a service from a third party	Self-maintained cluster of EC2 instances, or as a service from a third party	Self-maintained cluster of EC2 instances, or as a service from a third party
Increase available storage while running.	Not necessary. The database grows automatically.	Add more EC2 instances.	Increase EBS volumes while running.	Add more EC2 instances.	Add more EC2 instances.

13.2 DynamoDB for developers

DynamoDB is a key-value store that organizes your data into tables. For example, you can have a table to store your users and another table to store tasks. The items contained in the table are identified by a primary key. An item could be a user or a task; think of an item as a row in a relational database. A table can also maintain secondary indexes for data lookup in addition to the primary key, which is also similar to relational databases. In this section, you'll look at these basic building blocks of DynamoDB, ending with a brief comparison of NoSQL databases.

13.2.1 Tables, items, and attributes

Each DynamoDB table has a name and organizes a collection of items. An *item* is a collection of attributes, and an *attribute* is a name-value pair. The attribute value can be scalar (number, string, binary, Boolean), multivalued (number set, string set, binary set), or a JSON document (object, array). Items in a table aren't required to have the same attributes; there is no enforced schema. Figure 13.2 demonstrates these terms.

You can create a table with the Management Console, CloudFormation, SDKs, or the CLI. The following example shows how you'd create a table with the CLI (don't try to run this command now—you'll create a table later in the chapter). The table is named app-entity and uses the `id` attribute as the primary key:

The primary key is a partition key using the id attribute.

Choose a name for your table, like app-entity.

The attribute named id is of type string.

```
$ aws dynamodb create-table --table-name app-entity \
➥ --attribute-definitions AttributeName=id,AttributeType=S \
➥ --key-schema AttributeName=id,KeyType=HASH \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

You'll learn about this in section 13.9.

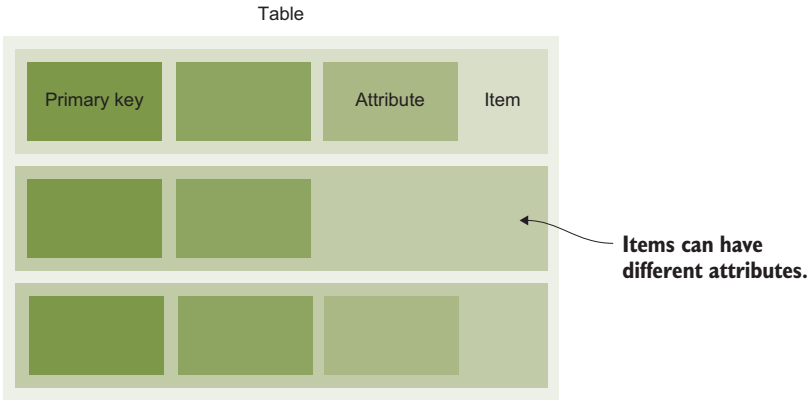


Figure 13.2 DynamoDB tables store items consisting of attributes identified by a primary key

If you plan to run multiple applications that use DynamoDB, it's good practice to prefix your tables with the name of your application. You can also add tables via the Management Console. *Keep in mind that you can't change the name of a table or its key schema later.* But you can add attribute definitions and change the throughput at any time.

13.2.2 Primary key

A primary key is unique within a table and identifies an item. You can use a single attribute as the primary key. DynamoDB calls this a *partition key*. You need an item's partition key to look up that item. You can also use two attributes as the primary key. In this case, one of the attributes is the partition key, and the other is called the *sort key*.

PARTITION KEY

A partition key uses a single attribute of an item to create a hash-based index. If you want to look up an item based on its partition key, you need to know the exact partition key. For example, a user table could use the user's email as a partition key. The user could then be retrieved if you know the partition key (email, in this case).

PARTITION KEY AND SORT KEY

When you use both a partition key and a sort key, you're using two attributes of an item to create a more powerful index. To look up an item, you need to know its exact partition key, but you don't need to know the sort key. You can even have multiple items with the same partition key: they will be sorted according to their sort key.

The partition key can only be queried using exact matches (=). The sort key can be queried using =, >, <, >=, <=, and BETWEEN x AND y operators. For example, you can query the sort key of a partition key from a certain starting point. You cannot query only the sort key—you must always specify the partition key. A message table could use a partition key and sort key as its primary key; the partition key could be the user's email, and the sort key could be a timestamp. You could then look up all of user's

messages that are newer or older than a specific timestamp, and the items would sorted according to the timestamp.

13.2.3 DynamoDB Local

Imagine a team of developers working on a new app using DynamoDB. During development, each developer needs an isolated database so as not to corrupt the other team members' data. They also want to write unit tests to make sure their app is working. You could create a unique set of DynamoDB tables with a CloudFormation stack for each developer. Or you could use a local DynamoDB for offline development. AWS provides an implementation of DynamoDB, which is available for download at http://mng.bz/27h5. Don't run it in production! It's only made for development purposes and provides the same functionality as DynamoDB, but it uses a different implementation: only the API is the same.

13.3 Programming a to-do application

To minimize the overhead of a programming language, you'll use Node.js/JavaScript to create a small to-do application that you can use via the terminal on your local machine. Let's call the application *nodetodo*. nodetodo will use DynamoDB as a database. With nodetodo, you can do the following:

- Create and delete users
- Create and delete tasks
- Mark tasks as done
- Get a list of all tasks with various filters

nodetodo supports multiple users and can track tasks with or without a due date. To help users deal with many tasks, tasks can be assigned to a category. nodetodo is accessed via the terminal. Here's how you would use nodetodo via the terminal to add a user. Important note: don't try to run the following commands now — they are not yet implemented. You will implement them in the following section.

Executes nodetodo in the terminal

Abstract description of the CLI: parameters are enclosed in < >.

```
# node index.js user-add <uid> <email> <phone>
$ node index.js user-add michael michael@widdix.de 0123456789
user added with uid michael
```

nodetodo's output is written to STDOUT.

To add a new task, you would do the following:

Named parameters are used with --name=value.

Optional parameters are enclosed in [ ].

```
# node index.js task-add <uid> <description> \
➥ [<category>] [--dueat=<yyyymmdd>]
$ node index.js task-add michael "plan lunch" --dueat=20150522
task added with tid 1432187491647
```

tid is the task ID.

You would mark a task as finished as follows:

```
# node index.js task-done <uid> <tid>
$ node index.js task-done michael 1432187491647
task completed with tid 1432187491647
```

You should also be able to list tasks. Here's how you would use nodetodo to do that:

```
# node index.js task-ls <uid> [<category>] [--overdue|--due|...]
$ node index.js task-ls michael
tasks [...]
```

To implement an intuitive CLI, nodetodo uses *docopt*, a command-line interface description language, to describe the CLI interface. The supported commands are as follows:

- `user-add`—Adds a new user to nodetodo
- `user-rm`—Removes a user
- `user-ls`—Lists users
- `user`—Shows the details of a single user
- `task-add`—Adds a new task to nodetodo
- `task-rm`—Removes a task
- `task-ls`—Lists user tasks with various filters
- `task-la`—Lists tasks by category with various filters
- `task-done`—Marks a task as finished

In the rest of the chapter, you'll implement those commands to learn about DynamoDB hands-on. This listing shows the full CLI description of all the commands, including parameters.

Listing 13.1 CLI description language docopt: using nodetodo (cli.txt)

```
nodetodo

Usage:
  nodetodo user-add <uid> <email> <phone>
  nodetodo user-rm <uid>
  nodetodo user-ls [--limit=<limit>] [--next=<id>]
  nodetodo user <uid>
  nodetodo task-add <uid> <description> \
➥ [<category>] [--dueat=<yyyymmdd>]
  nodetodo task-rm <uid> <tid>
  nodetodo task-ls <uid> [<category>] \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-la <category> \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-done <uid> <tid>
  nodetodo -h | --help
  nodetodo --version
```

The named parameters limit and next are optional.

The category parameter is optional.

Pipe indicates either/or.

help prints information about how to use nodetodo.

Version information

```
Options:
  -h --help         Show this screen.
  --version         Show version.
```

DynamoDB isn't comparable to a traditional relational database in which you create, read, update, or delete data with SQL. You'll access DynamoDB with an SDK to call the HTTPS REST API. You must integrate DynamoDB into your application; you can't take an existing application that uses an SQL database and run it on DynamoDB. To use DynamoDB, you need to write code!

13.4 Creating tables

Tables in DynamoDB organize your data. You aren't required to define all the attributes that the table items will have. DynamoDB doesn't need a static schema like a relational database does, but you must define the attributes that are used as the primary key in your table. In other words, you must define the table's primary key schema. To do so, you'll use the AWS CLI. The `aws dynamodb create-table` command has four mandatory options:

- `table-name`—Name of the table (can't be changed).
- `attribute-definitions`—Name and type of attributes used as the primary key. Multiple definitions can be given using the syntax `AttributeName=attr1, AttributeType=S`, separated by a space character. Valid types are `S` (String), `N` (Number), and `B` (Binary).
- `key-schema`—Name of attributes that are part of the primary key (can't be changed). Contains a single entry using the syntax `AttributeName=attr1,KeyType=HASH` for a partition key, or two entries separated by spaces for a partition key and sort key. Valid types are `HASH` and `RANGE`.
- `provisioned-throughput`—Performance settings for this table defined as `ReadCapacityUnits=5,WriteCapacityUnits=5` (you'll learn about this in section 13.9).

You'll now create a table for the users of the nodetodo application as well as a table that will contain all the tasks.

13.4.1 Users are identified by a partition key

Before you create a table for nodetodo users, you must think carefully about the table's name and primary key. We suggest that you prefix all your tables with the name of your application. In this case, the table name would be `todo-user`. To choose a primary key, you have to think about the queries you'll make in the future and whether there is something unique about your data items. Users will have a unique ID, called `uid`, so it makes sense to choose the `uid` attribute as the partition key. You must also be able to look up users based on the `uid` to implement the `user` command. Use a single attribute as primary key by marking the attribute as the partition key of your table. The following

example shows a user table where the attribute `uid` is used as the partition key of the primary key:

```
"michael" => {                    <-- uid ("michael") is the partition key; everything in { } is the item.
  "uid": "michael",
  "email": "michael@widdix.de",
  "phone": "0123456789"
}
"andreas" => {                    <-- Partition keys have no order.
  "uid": "andreas",
  "email": "andreas@widdix.de",
  "phone": "0123456789"
}
```

Because users will only be looked up based on the known `uid`, it's fine to use a partition key to identify a user. Next you'll create the user table, structured like the previous example, with the help of the AWS CLI:

```
$ aws dynamodb create-table --table-name todo-user \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ --key-schema AttributeName=uid,KeyType=HASH \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

Prefixing tables with the name of your application will prevent name clashes in the future.

Items must at least have one attribute uid of type string.

You'll learn about this in section 13.9.

The partition key (type HASH) uses the uid attribute.

Creating a table takes some time. Wait until the status changes to `ACTIVE`. You can check the status of a table as follows:

```
$ aws dynamodb describe-table --table-name todo-user
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "uid",
        "AttributeType": "S"
      }
    ],
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "WriteCapacityUnits": 5,
      "ReadCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "TableName": "todo-user",
    "TableStatus": "ACTIVE",
    "KeySchema": [
      {
        "KeyType": "HASH",
        "AttributeName": "uid"
      }
```

CLI command to check the table status

Attributes defined for that table

Status of the table

Attributes used as the primary key

```
    ],
    "ItemCount": 0,
    "CreationDateTime": 1432146267.678
  }
}
```

13.4.2 Tasks are identified by a partition key and sort key

Tasks always belong to a user, and all commands that are related to tasks include the user's ID. To implement the `task-ls` command, you need a way to query the tasks based on the user's ID. In addition to the partition key, you can use the sort key. This way, you can add multiple items to the same partition key. Because all interactions with tasks require the user's ID, you can choose `uid` as the partition key and a task ID (`tid`), the timestamp of creation, as the sort key. Now you can make queries that include the user's ID and, if needed, the task's ID.

> **NOTE** This solution has one limitation: users can add only one task per timestamp. Because tasks are uniquely identified by `uid` and `tid` (the primary key) there can't be two tasks for the same user at the same time. Our timestamp comes with millisecond resolution, so it should be fine.

Using a partition key and a sort key uses two of your table attributes. For the partition key, an unordered hash index is maintained; the sort key is kept in a sorted index for each partition key. The combination of the partition key and the sort key uniquely identifies an item if they are used as the primary key. The following data set shows the combination of unsorted partition keys and sorted sort keys:

```
["michael", 1] => {
  "uid": "michael",
  "tid": 1,
  "description": "prepare lunch"
}
["michael", 2] => {
  "uid": "michael",
  "tid": 2,
  "description": "buy nice flowers for mum"
}
["michael", 3] => {
  "uid": "michael",
  "tid": 3,
  "description": "prepare talk for conference"
}
["andreas", 1] => {
  "uid": "andreas",
  "tid": 1,
  "description": "prepare customer presentation"
}
["andreas", 2] => {
  "uid": "andreas",
  "tid": 2,
  "description": "plan holidays"
}
```

uid ("michael") is the partition key and tid (1) is the sort key of the primary key.

The sort keys are sorted within a partition key.

There is no order in the partition keys.

nodetodo offers the ability to get all tasks for a user. If the tasks have only a partition key as the primary key, this will be difficult, because you need to know the partition key to extract them from DynamoDB. Luckily, using the partition key and sort key as the primary key makes things easier, because you only need to know the partition key to extract the items. For the tasks, you'll use `uid` as the known partition key. The sort key is `tid`. The task ID is defined as the timestamp when the task was created. You'll now create the task table, using two attributes to create a partition key and sort key as the primary key:

At least two attributes are needed for a partition key and sort key.

```
$ aws dynamodb create-table --table-name todo-task \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ --key-schema AttributeName=uid,KeyType=HASH \
➥ AttributeName=tid,KeyType=RANGE \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

The tid attribute is the sort key.

Wait until the table status changes to `ACTIVE` when you run `aws dynamodb describe-table --table-name todo-task`. When both tables are ready, you'll add data.

13.5 Adding data

You have two tables up and running to store users and their tasks. To use them, you need to add data. You'll access DynamoDB via the Node.js SDK, so it's time to set up the SDK and some boilerplate code before you implement adding users and tasks.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.

After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v8.*`. Now you're ready to run JavaScript examples like nodetodo for AWS.

Do you want to get started with Node.js? We recommend *Node.js in Action* (Second Edition) by Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* by P.J. Evans, (Manning, 2018).

To get started with Node.js and docopt, you need some magic lines to load all the dependencies and do some configuration work. Listing 13.2 shows how this can be done.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. nodetodo is located in /chapter13/. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

Docopt is responsible for reading all the arguments passed to the process. It returns a JavaScript object, where the arguments are mapped to the parameters in the CLI description.

Listing 13.2 nodetodo: using docopt in Node.js (index.js)

```
const fs = require('fs');
const docopt = require('docopt');
const moment = require('moment');
const AWS = require('aws-sdk');
const db = new AWS.DynamoDB({
  region: 'us-east-1'
});

const cli = fs.readFileSync('./cli.txt', {encoding: 'utf8'});
const input = docopt.docopt(cli, {
  version: '1.0',
  argv: process.argv.splice(2)
});
```


Next you'll implement the features of nodetodo. You can use the `putItem` SDK operation to add data to DynamoDB like this:

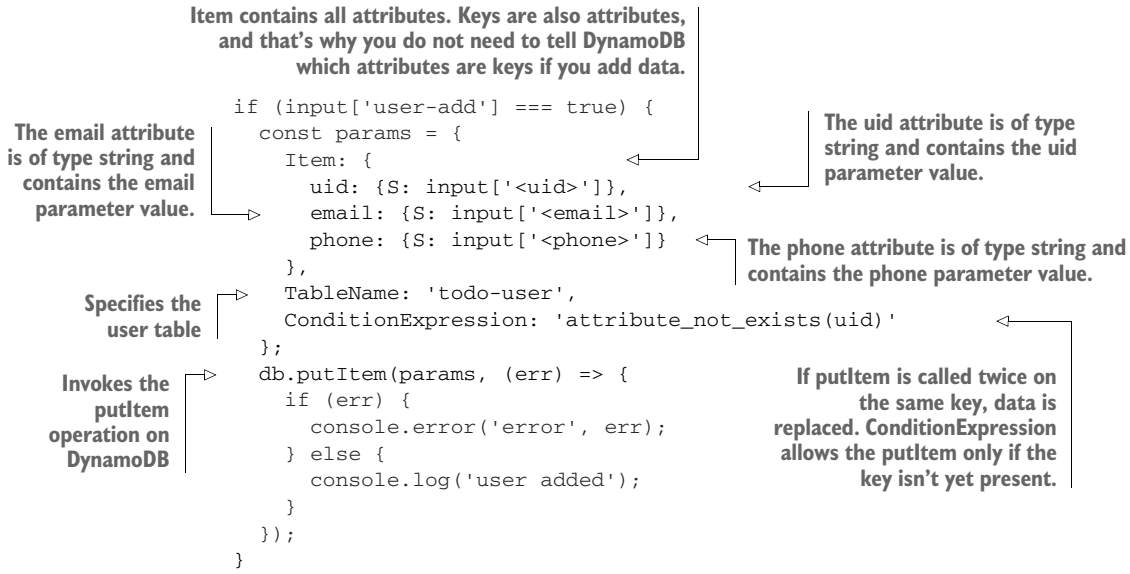
```
const params = {
  Item: {
    attr1: {S: 'val1'},
    attr2: {N: '2'}
  },
  TableName: 'app-entity'
};
db.putItem(params, (err) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('success');
  }
});
```


The first step is to add data to nodetodo.

13.5.1 Adding a user

You can add a user to nodetodo by calling `nodetodo user-add <uid> <email> <phone>`. In Node.js, you do this using this code.

Listing 13.3 nodetodo: adding a user (index.js)




```
if (input['user-add'] === true) {
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      email: {S: input['<email>']},
      phone: {S: input['<phone>']}
    },
    TableName: 'todo-user',
    ConditionExpression: 'attribute_not_exists(uid)'
  };
  db.putItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('user added');
    }
  });
}
```

When you make a call to the AWS API, you always do the following:

1. Create a JavaScript object (map) filled with the needed parameters (the params variable).
2. Invoke the function on the AWS SDK.
3. Check whether the response contains an error, and if not, process the returned data.

Therefore you only need to change the content of `params` if you want to add a task instead of a user.

13.5.2 Adding a task

You can add a task to nodetodo by calling `nodetodo task-add <uid> <description> [<category>] [--dueat=<yyyymmdd>]`. For example, to create a task to remember the milk, you could add a task like this: `nodetodo task-add michael "buy milk"`. In Node.js, you can implement this command with the code in listing 13.4.

Listing 13.4 nodetodo: adding a task (index.js)

```
if (input['task-add'] === true) {
  const tid = Date.now();
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      tid: {N: tid.toString()},
      description: {S: input['<description>']},
      created: {N: moment(tid).format('YYYYMMDD')}
    },
    TableName: 'todo-task',
    ConditionExpression: 'attribute_not_exists(uid)
  and attribute_not_exists(tid)'
  };
  if (input['--dueat'] !== null) {
    params.Item.due = {N: input['--dueat']};
  }
  if (input['<category>'] !== null) {
    params.Item.category = {S: input['<category>']};
  }
  db.putItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task added with tid ' + tid);
    }
  });
}
```

Creates the task ID (tid) based on the current timestamp

The tid attribute is of type number and contains the tid value.

The created attribute is of type number (format 20150525).

Specifies the task table

If the optional named parameter dueat is set, add this value to the item.

Ensures that an existing item is not overridden

If the optional named parameter category is set, add this value to the item.

Invokes the putItem operation on DynamoDB

Now you can add users and tasks to nodetodo. Wouldn't it be nice if you could retrieve all this data?

13.6 Retrieving data

DynamoDB is a key-value store. The key is usually the only way to retrieve data from such a store. When designing a data model for DynamoDB, you must be aware of that limitation when you create tables (as you did in section 13.4). If you can use only one key to look up data, you'll sooner or later experience difficulties. Luckily, DynamoDB provides two other ways to look up items: a secondary index key lookup, and the scan operation. You'll start by retrieving data with its primary key and continue with more sophisticated methods of data retrieval.

DynamoDB Streams

DynamoDB lets you retrieve changes to a table as soon as they're made. A *stream* provides all write (create, update, delete) operations to your table items. The order is consistent within a partition key:

- If your application polls the database for changes, DynamoDB Streams solves the problem in a more elegant way.

- If you want to populate a cache with the changes made to a table, DynamoDB Streams can help.
- If you want to replicate a DynamoDB table to another region, DynamoDB Streams can do it.

13.6.1 Getting an item by key

The simplest form of data retrieval is looking up a single item by its primary key, for example a user by its ID. The `getItem` SDK operation to get a single item from DynamoDB can be used like this:

```
const params = {
  Key: {
    attr1: {S: 'val1'}
  },
  TableName: 'app-entity'
};
db.getItem(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    if (data.Item) {
      console.log('item', data.Item);
    } else {
      console.error('no item found');
    }
  }
});
```

Specifies the attributes of the primary key

Invokes the getItem operation on DynamoDB

Checks whether an item was found

The command `nodetodo user <uid>` must retrieve a user by the user's ID (uid). Translated to the Node.js AWS SDK, this looks like the following listing.

Listing 13.5 nodetodo: retrieving a user (index.js)

```
const mapUserItem = (item) => {
  return {
    uid: item.uid.S,
    email: item.email.S,
    phone: item.phone.S
  };
};

if (input['user'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}
    },
    TableName: 'todo-user'
  };
```

Helper function to transform DynamoDB result

Looks up a user by primary key

Specifies the user table

```
  db.getItem(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      if (data.Item) {
        console.log('user', mapUserItem(data.Item));
      } else {
        console.error('user not found');
      }
    }
  });
}
```

Invokes the getItem operation on DynamoDB

Checks whether data was found for the primary key

You can also use the `getItem` operation to retrieve data by partition key and sort key, for example to look up a specific task. The only change is that the `Key` has two entries instead of one. `getItem` returns one item or no items; if you want to get multiple items, you need to query DynamoDB.

13.6.2 Querying items by key and filter

If you want to retrieve a collection of items rather than a single item, such as all tasks for a user, you must query DynamoDB. Retrieving multiple items by primary key only works if your table has a partition key and sort key. Otherwise, the partition key will only identify a single item. The `query` SDK operation to get a collection of items from DynamoDB can be used like this:

```
const params = {
  KeyConditionExpression: 'attr1 = :attr1val AND attr2 = :attr2val',
  ExpressionAttributeValues: {
    ':attr1val': {S: 'val1'},
    ':attr2val': {N: '2'}
  },
  TableName: 'app-entity'
};
db.query(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
  }
});
```

Condition the key must match. Use AND if you're querying both a partition and sort key. Only the = operator is allowed for partition keys. Allowed operators for sort keys: =, >, <, >=, <=, BETWEEN x AND y, and begins_with. Sort key operators are blazing fast because the data is already sorted.

Dynamic values are referenced in the expression.

Always specify the correct type (S, N, B).

Invokes the query operation on DynamoDB

The `query` operation also lets you specify an optional `FilterExpression`, to include only items that match the filter and key condition. This is helpful to reduce the result set, for example to show only tasks of a specific category. The syntax of `FilterExpression` works like `KeyConditionExpression`, but no index is used for filters. Filters are applied to all matches that `KeyConditionExpression` returns.

To list all tasks for a certain user, you must query DynamoDB. The primary key of a task is the combination of the `uid` and the `tid`. To get all tasks for a user, `KeyConditionExpression` only requires the partition key. The implementation of `nodetodo task-ls <uid> [<category>] [--overdue|--due|--withoutdue|--futuredue]` is shown next.

Listing 13.6 nodetodo: retrieving tasks (index.js)

```
const getValue = (attribute, type) => {
  if (attribute === undefined) {
    return null;
  }
  return attribute[type];
};

const mapTaskItem = (item) => {
  return {
    tid: item.tid.N,
    description: item.description.S,
    created: item.created.N,
    due: getValue(item.due, 'N'),
    category: getValue(item.category, 'S'),
    completed: getValue(item.completed, 'N')
  };
};

if (input['task-ls'] === true) {
  const params = {
    KeyConditionExpression: 'uid = :uid',
    ExpressionAttributeValues: {
      ':uid': {S: input['<uid>']}
    },
    TableName: 'todo-task',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression +=
      ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--due'] === true) {
    params.FilterExpression = 'due = :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--withoutdue'] === true) {
    params.FilterExpression = 'attribute_not_exists(due)';
  } else if (input['--futuredue'] === true) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--dueafter'] !== null) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
```

Helper function to access optional attributes

Helper function to transform the DynamoDB result

Primary key query. The task table uses a partition and sort key. Only the partition key is defined in the query, so all tasks belonging to a user are returned.

Query attributes must be passed this way.

Filtering uses no index; it's applied over all elements returned from the primary key query.

Filter attributes must be passed this way.

attribute_not_exists(due) is true when the attribute is missing (opposite of attribute_exists).

```
      {N: input['--dueafter']};
  } else if (input['--duebefore'] !== null) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
      {N: input['--duebefore']};
  }
  if (input['<category>'] !== null) {
    if (params.FilterExpression === undefined) {
      params.FilterExpression = '';
    } else {
      params.FilterExpression += ' AND ';
    }
    params.FilterExpression += 'category = :category';
    params.ExpressionAttributeValues[':category'] =
      S: input['<category>']};
  }
  db.query(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next=' +
          data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

Multiple filters can be combined with logical operators.

Invokes the query operation on DynamoDB

Two problems arise with the query approach:

1. Depending on the result size from the primary key query, filtering may be slow. Filters work without an index: every item must be inspected. Imagine you have stock prices in DynamoDB, with a partition key and sort key: the partition key is a ticker like AAPL, and the sort key is a timestamp. You can make a query to retrieve all stock prices of Apple (AAPL) between two timestamps (20100101 and 20150101). But if you only want to return prices on Mondays, you need to filter over all prices to return only 20% (1 out of 5 trading days each week) of them. That's wasting a lot of resources!
2. You can only query the primary key. Returning a list of all tasks that belong to a certain category for all users isn't possible, because you can't query the `category` attribute.

You can solve those problems with secondary indexes. Let's look at how they work.

13.6.3 Using global secondary indexes for more flexible queries

A *global secondary index* is a projection of your original table that is automatically maintained by DynamoDB. Items in an index don't have a primary key, just a key. This key is not necessarily unique within the index. Imagine a table of users where each user has a country attribute. You then create a global secondary index where the country is

the new partition key. As you can see, many users can live in the same country, so that key is not unique in the index.

You can query a global secondary index like you would query the table. You can imagine a global secondary index as a read-only DynamoDB table that is automatically maintained by DynamoDB: whenever you change the parent table, all indexes are asynchronously (eventually consistent!) updated as well. Figure 13.3 shows how a global secondary index works.

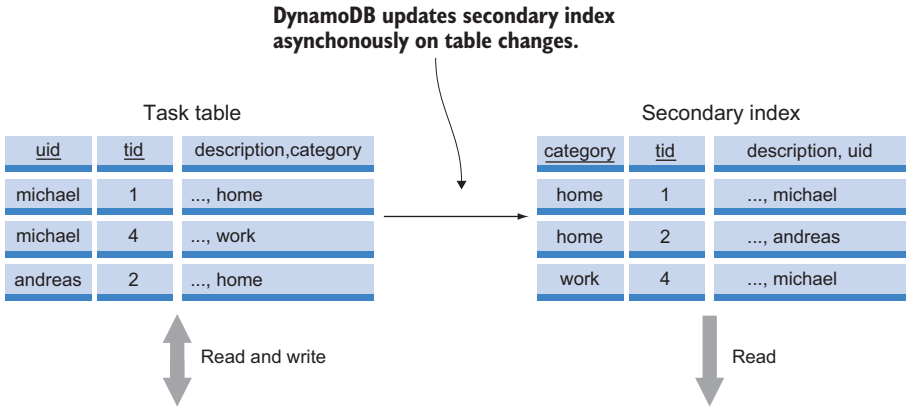


Figure 13.3 A global secondary index contains a copy (projection) of your table's data to provide fast lookup on another key.

A global secondary index comes at a price: the index requires storage (the same cost as for the original table). You must provision additional write-capacity units for the index as well, because a write to your table will cause a write to the global secondary index as well.

Local secondary index

Besides global secondary indexes, DynamoDB also support local secondary indexes. A local secondary index must use the same partition key as the table. You can only vary on the attribute that is used as the sort key. A local secondary index uses the read and write-capacity of the table.

A huge benefit of DynamoDB is that you can provision capacity based on your workload. If one of your global secondary indexes gets tons of read traffic, you can increase the read capacity of that index. You can fine-tune your database performance by provisioning sufficient capacity for your tables and indexes. You'll learn more about that in section 13.9.

Back to nodetodo. To implement the retrieval of tasks by category, you'll add a secondary index to the `todo-task` table. This will allow you to make queries by category.

A partition key and sort key are used: the partition key is the `category` attribute, and the sort key is the `tid` attribute. The index also needs a name: `category-index`. You can find the following CLI command in the README.md file in nodetodo's code folder:

```
$ aws dynamodb update-table --table-name todo-task \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ AttributeName=category,AttributeType=S \
➥ --global-secondary-index-updates '[{\
➥ "Create": {\
➥ "IndexName": "category-index", \
➥ "KeySchema": [{"AttributeName": "category", "KeyType": "HASH"}, \
➥ {"AttributeName": "tid", "KeyType": "RANGE"}], \
➥ "Projection": {"ProjectionType": "ALL"}, \
➥ "ProvisionedThroughput": {"ReadCapacityUnits": 5, \
➥ "WriteCapacityUnits": 5}\
➥ }}]'
```

You can add a global secondary index after the table is created.

Adds a category attribute, because the attribute will be used in the index.

Creates a new secondary index

The category attribute is the partition key, and the tid attribute is the sort key.

All attributes are projected into the index.

Creating a global secondary index takes some time. You can use the CLI to find out if the index is ready:

```
$ aws dynamodb describe-table --table-name=todo-task \
➥ --query "Table.GlobalSecondaryIndexes"
```

The following listing shows how the implementation of `nodetodo task-la <category> [--overdue|...]` uses the `query` operation.

Listing 13.7 nodetodo: retrieving tasks from a category index (index.js)

```
if (input['task-la'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    KeyConditionExpression: 'category = :category',
    ExpressionAttributeValues: {
      ':category': {S: input['<category>']}
    },
    TableName: 'todo-task',
    IndexName: 'category-index',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression += ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  }
  [...]
  db.query(params, (err, data) => {
```

A query against an index works the same as a query against a table…

…but you must specify the index you want to use.

Filtering works the same as with tables

```
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next='
          + data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

But there are still situations where a query doesn't work: you can't retrieve all users. Let's look at what a table `scan` can do for you.

13.6.4 Scanning and filtering all of your table's data

Sometime you can't work with keys because you don't know them up front; instead, you need to go through all the items in the table. That's not very efficient, but in rare situations like daily batch jobs, it's okay. DynamoDB provides the `scan` operation to scan all items in a table:

```
const params = {
  TableName: 'app-entity',
  Limit: 50
};
db.scan(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
    if (data.LastEvaluatedKey !== undefined) {
      console.log('more items available');
    }
  }
});
```

Specifies the maximum number of items to return

Invokes the scan operation on DynamoDB

Checks whether there are more items that can be scanned

The next listing shows the implementation of `nodetodo user-ls [--limit=<limit>] [--next=<id>]`. A paging mechanism is used to prevent too many items from being returned.

Listing 13.8 nodetodo: retrieving all users with paging (index.js)

```
if (input['user-ls'] === true) {
  const params = {
    TableName: 'todo-user',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.ExclusiveStartKey = {
      uid: {S: input['--next']}
    };
```

Maximum number of items returned

The named parameter next contains the last evaluated key.

```
  }
  db.scan(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('users', data.Items.map(mapUserItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('page with --next=' + data.LastEvaluatedKey.uid.S);
      }
    }
  });
}
```

Invokes the scan operation on DynamoDB

Checks whether the last item has been reached

The `scan` operation reads all items in the table. This example didn't filter any data, but you can use `FilterExpression` as well. Note that you shouldn't use the `scan` operation too often—it's flexible but not efficient.

13.6.5 Eventually consistent data retrieval

DynamoDB doesn't support transactions the same way a traditional database does. You can't modify (create, update, delete) multiple documents in a single transaction—the atomic unit in DynamoDB is a single item (to be more precise, a partition key).

In addition, DynamoDB is eventually consistent. That means it's possible that if you create an item (version 1), update that item to version 2, and then get that item, you may see the old version 1; if you wait and get the item again, you'll see version 2. Figure 13.4 shows this process. The reason for this behavior is because the item is persisted on multiple machines in the background. Depending on which machine answers your request, the machine may not have the latest version of the item.

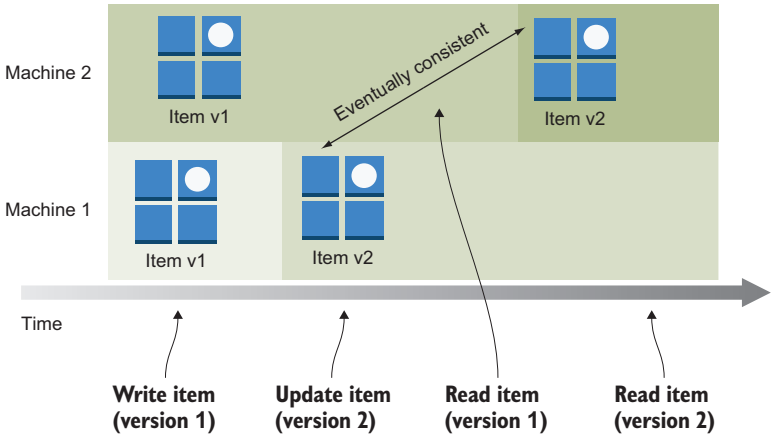


Figure 13.4 Eventually consistent reads can return old values after a write operation until the change is propagated to all machines.

You can prevent eventually consistent reads by adding `"ConsistentRead": true` to the DynamoDB request to get *strongly consistent reads*. Strongly consistent reads are supported by `getItem`, `query`, and `scan` operations. But a strongly consistent read takes longer and consumes more read capacity than an eventually consistent read. Reads from a global secondary index are always eventually consistent because the index itself is eventually consistent.

13.7 Removing data

Like the `getItem` operation, the `deleteItem` operation requires that you specify the primary key you want to delete. Depending on whether your table uses a partition key or a partition key and sort key, you must specify one or two attributes.

You can remove a user with nodetodo by calling `nodetodo user-rm <uid>`. In Node.js, you do this as shown in the following listing.

Listing 13.9 nodetodo: removing a user (index.js)

```
if (input['user-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}
    },
    TableName: 'todo-user'
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('user removed');
    }
  });
}
```

Identifies an item by partition key

Specifies the user table

Invokes the deleteItem operation on DynamoDB

Removing a task is similar: `nodetodo task-rm <uid> <tid>`. The only change is that the item is identified by a partition key and sort key, and the table name has to be changed.

Listing 13.10 nodetodo: removing a task (index.js)

```
if (input['task-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}
    },
    TableName: 'todo-task'
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
```

Identifies an item by partition key and sort key

Specifies the task table

```
    } else {
      console.log('task removed');
    }
  });
}
```

You're now able to create, read, and delete items in DynamoDB. The only operation you're missing is updating.

13.8 Modifying data

You can update an item with the `updateItem` operation. You must identify the item you want to update by its primary key; you can also provide an `UpdateExpression` to specify the updates you want to perform. You can use one or a combination of the following update actions:

- Use `SET` to override or create a new attribute. Examples: `SET attr1 = :attr1val`, `SET attr1 = attr2 + :attr2val`, `SET attr1 = :attr1val, attr2 = :attr2val`.
- Use `REMOVE` to remove an attribute. Examples: `REMOVE attr1`, `REMOVE attr1, attr2`.

In nodetodo, you can mark a task as done by calling `nodetodo task-done <uid> <tid>`. To implement this feature, you need to update the task item, as shown next in Node.js.

Listing 13.11 nodetodo: updating a task as done (index.js)

```
if (input['task-done'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}
    },
    UpdateExpression: 'SET completed = :yyyymmdd',
    ExpressionAttributeValues: {
      ':yyyymmdd': {N: yyyymmdd}
    },
    TableName: 'todo-task'
  };
  db.updateItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task completed');
    }
  });
}
```

Identifies the item by a partition and sort key

Defines which attributes should be updated

Attribute values must be passed this way.

Invokes the updateItem operation on DynamoDB

Cleaning up

Don't forget to delete your DynamoDB tables after you finish this section. Use the Management Console to do so.

That's it! You've implemented all of nodetodo's features.

13.9 Scaling capacity

When you create a DynamoDB table or a global secondary index, you must provision throughput. Throughput is divided into read and write capacity. DynamoDB uses `ReadCapacityUnits` and `WriteCapacityUnits` to specify the throughput of a table or global secondary index. But how is a capacity unit defined?

13.9.1 Capacity units

To understand capacity units, let's start by experimenting with the CLI:

```
$ aws dynamodb get-item --table-name todo-user \
➥ --key '{"uid": {"S": "michael"}}' \
➥ --return-consumed-capacity TOTAL \
➥ --query "ConsumedCapacity"
{
    "CapacityUnits": 0.5,
    "TableName": "todo-user"
}
```

Tells DynamoDB to return the used capacity units

getItem requires 0.5 capacity units.

```
$ aws dynamodb get-item --table-name todo-user \
➥ --key '{"uid": {"S": "michael"}}' \
➥ --consistent-read --return-consumed-capacity TOTAL \
➥ --query "ConsumedCapacity"
{
    "CapacityUnits": 1.0,
    "TableName": "todo-user"
}
```

A consistent read…

…needs twice as many capacity units.

More abstract rules for throughput consumption are as follows:

- An eventually consistent read takes half the capacity of a strongly consistent read.
- A strongly consistent `getItem` requires one read capacity unit if the item isn't larger than 4 KB. If the item is larger than 4 KB, you need additional read capacity units. You can calculate the required read capacity units using `roundUP(itemSize / 4)`.
- A strongly consistent `query` requires one read capacity unit per 4 KB of item size. This means if your query returns 10 items, and each item is 2 KB, the item size is 20 KB and you need 5 read units. This is in contrast to 10 `getItem` operations, for which you would need 10 read capacity units.

- A write operation needs one write capacity unit per 1 KB of item size. If your item is larger than 1 KB, you can calculate the required write capacity units using `roundUP(itemSize)`.

If capacity units aren't your favorite unit, you can use the AWS Simple Monthly Calculator at http://aws.amazon.com/calculator to calculate your capacity needs by providing details of your read and write workload.

The provision throughput of a table or a global secondary index is defined in seconds. If you provision five read capacity units per second with `ReadCapacityUnits=5`, you can make five strongly consistent `getItem` requests for that table if the item size isn't larger than 4 KB per second. If you make more requests than are provisioned, DynamoDB will first throttle your request. If you make many more requests than are provisioned, DynamoDB will reject your requests.

It's important to monitor how many read and write capacity units you require. Fortunately, DynamoDB sends some useful metrics to CloudWatch every minute. To see the metrics, open the AWS Management Console, navigate to the DynamoDB service, and select one of the tables ❶ and click the Metrics tab ❷. Figure 13.5 shows the CloudFormation metrics for the `todo-user` table.

Increasing the provisioned throughput is possible whenever you like, but you can only decrease the throughput of a table four to nine times a day (a day in UTC time). Therefore, you might need to overprovision the throughput of a table during some times of the day.

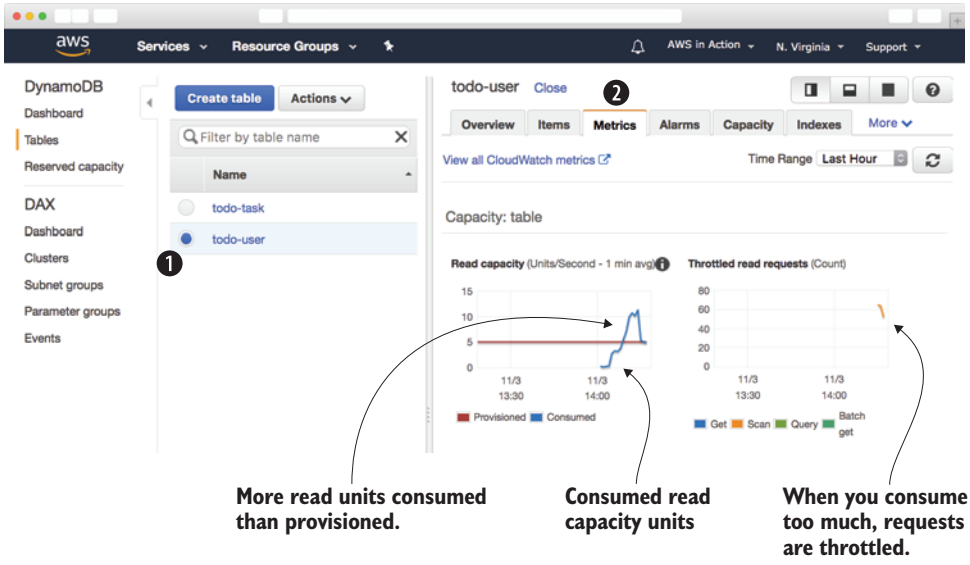


Figure 13.5 Monitoring provisioned and consumed capacity units of the DynamoDB table

Limits for decreasing the throughput capacity

Decreasing the throughput capacity of a table is generally only allowed four times a day (day in UTC time). Additionally, decreasing the throughout capacity is possible even if you have used up all four decreases in case the last decrease has happened more than four hours ago.

Theoretically, you can decrease the throughput capacity of your table up to nine times a day. When decreasing capacity four times in the first hour of the day, you get an additional decrease in the 5th hour. Next, after a decrease in the 5th hour, an additional decrease in the 11th hour is possible, and so on.

13.9.2 Auto-scaling

You can adjust the capacity of your DynamoDB tables and global secondary indexes based on your database's load. If your application tier scales automatically, it's a good idea to scale the database as well. Otherwise your database will become the bottleneck. The service used to implement this is called *Application Auto Scaling*. The following CloudFormation snippet shows how you can define auto-scaling rules:

```
# [...]
RoleScaling:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: 2012-10-17
      Statement:
      - Effect: Allow
        Principal:
          Service: 'application-autoscaling.amazonaws.com'
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: scaling
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Action:
          - 'dynamodb:DescribeTable'
          - 'dynamodb:UpdateTable'
          - 'cloudwatch:PutMetricAlarm'
          - 'cloudwatch:DescribeAlarms'
          - 'cloudwatch:DeleteAlarms'
          Resource: '*'
TableWriteScalableTarget:
  Type: 'AWS::ApplicationAutoScaling::ScalableTarget'
  Properties:
    MaxCapacity: 20
    MinCapacity: 5
    ResourceId: 'table/todo-user'
    RoleARN: !GetAtt 'RoleScaling.Arn'
```

The IAM role is needed to allow AWS to adjust your tables.

Not more than 20 capacity units…

…not less than 5 capacity units…

…reference the DynamoDB table.

```
      ScalableDimension: 'dynamodb:table:WriteCapacityUnits'
      ServiceNamespace: dynamodb
  TableWriteScalingPolicy:
    Type: 'AWS::ApplicationAutoScaling::ScalingPolicy'
    Properties:
      PolicyName: TableWriteScalingPolicy
      PolicyType: TargetTrackingScaling
      ScalingTargetId: !Ref TableWriteScalableTarget
      TargetTrackingScalingPolicyConfiguration:
        TargetValue: 50.0
        ScaleInCooldown: 600
        ScaleOutCooldown: 60
        PredefinedMetricSpecification:
          PredefinedMetricType: DynamoDBWriteCapacityUtilization
```

Scale is write capacity units; you can also choose dynamodb:table:ReadCapacityUnits.

Wait at least 600 seconds between two scale ins (decrease in capacity).

Adjust capacity to reach an target utilization of 50%.

Wait at least 60 seconds between two scale outs (increase in capacity).

To help you explore DynamoDB auto-scaling, we created a CloudFormation template, located at http://mng.bz/3S89. Create a stack based on that template by clicking on the CloudFormation Quick-Create link at http://mng.bz/dsRI, and it will create the tables needed for the nodetodo application, including auto-scaling.

Figure 13.6 shows auto-scaling in action. At 18:10, the capacity is no longer sufficient. Therefore the capacity is automatically increased at 18:30.

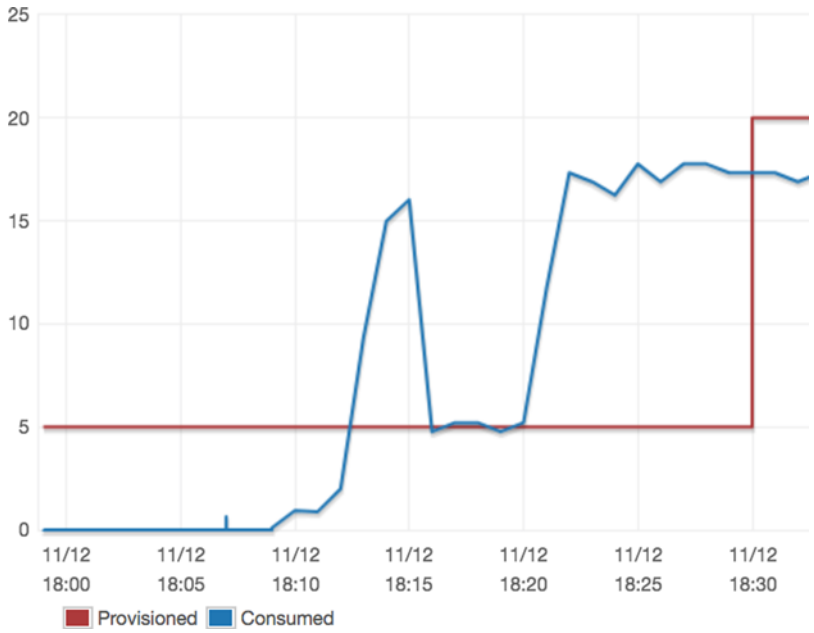


Figure 13.6 DynamoDB read capacity auto-scaling at work

Combine this knowledge of a database that scales with chapter 17, where you will learn to scale a fleet of EC2 instances. DynamoDB is the only database on AWS that grows and shrinks with your load.

Cleaning up

Don't forget to delete the CloudFormation stack `nodetodo`. Use the Management Console to do so.

Summary

- DynamoDB is a NoSQL database service that removes all the operational burdens from you, scales well, and can be used in many ways as the storage back end of your applications.
- Looking up data in DynamoDB is based on keys. You can only look up a partition key if you know the exact key. But DynamoDB also supports using a partition key and sort key, which combines the power of a partition key with another key that is sorted and supports range queries.
- You can enforce strongly consistent reads to avoid running into eventual consistency issues with stale data. But reads from a global secondary index are always eventually consistent.
- DynamoDB doesn't support SQL. Instead, you must use the SDK to communicate with DynamoDB from your application. This also implies that you can't use an existing application to run with DynamoDB without touching the code.
- Monitoring consumed read and write capacity is important if you want to provision enough capacity for your tables and indices.
- DynamoDB is charged per gigabyte of storage and per provisioned read or write capacity.
- You can use the `query` operation to query table or secondary indexes.
- The `scan` operation is flexible but not efficient and shouldn't be used too often.

Part 4

Architecting on AWS

Werner Vogels, CTO of Amazon.com, is quoted as saying "Everything fails all the time." Instead of trying to reach the unreachable goal of an unbreakable system, AWS plans for failure:

- Hard drives can fail, so S3 stores data on multiple hard drives to prevent loss of data.
- Computing hardware can fail, so virtual machines can be automatically restarted on another machine if necessary.
- Data centers can fail, so there are multiple data centers per region that can be used in parallel or on demand.

Outages of IT infrastructure and applications can cause loss of trust and money, and are a major risk for business. You will learn how to prevent an outage of your AWS applications by using the right tools and architecture.

Some AWS services handle failure by default in the background. For some services, responding to failure scenarios is available on demand. And some services don't handle failure by themselves, but offer the possibility to plan and react to failure. The following table shows an overview of the most important services and their failure handling.

Designing for failure is a fundamental principle on AWS. Another one is to make use of the elasticity of the cloud. You will learn about how to increase the number of your virtual machines based on the current workload. This will allow you to architect reliable systems for AWS.

Overview of services and their failure handling possibilities

	Description	Examples
Fault tolerant	Services can recover from failure automatically without any downtime.	S3 (object storage), DynamoDB (NoSQL database), Route 53 (DNS)
Highly available	Services can recover from some failures with a small downtime automatically.	RDS (relational database), EBS (network attached storage)
Manual failure handling	Services do not recover from failure by default but offer tools to build a highly available infrastructure on top of them.	EC2 (virtual machine)

Chapter 14 lays the foundation for becoming independent of the risk of losing a single server or a complete data center. You will learn how to recover a single EC2 instance either in the same data center or in another data center.

Chapter 15 introduces the concept of decoupling your system to increase reliability. You will learn how to use synchronous decoupling with the help of load balancers on AWS. You'll also see asynchronous decoupling by using Amazon SQS, a distributed queuing service, to build a fault-tolerant system.

Chapter 16 uses a lot of the services you've discovered so far to built a fault-tolerant application. You'll learn everything you need to design a fault-tolerant web application based on EC2 instances (which aren't fault-tolerant by default).

Chapter 17 is all about elasticity. You will learn how to scale your capacity based on a schedule, or based on the current load of your system.

14 Achieving high availability: availability zones, auto-scaling, and CloudWatch

This chapter covers

- Using a CloudWatch alarm to recover a failed virtual machine
- Understanding availability zones in an AWS region
- Using auto-scaling to guarantee your VMs keep running
- Analyzing disaster-recovery requirements

Imagine you run a web shop. During the night, the hardware running your virtual machine fails. Until the next morning when you go into work, your users can no longer access your web shop. During the 8-hour downtime, your users search for an alternative and stop buying from you. That's a disaster for any business. Now imagine a highly available web shop. Just a few minutes after the hardware failed, the system recovers, restarts itself on new hardware, and your web shop is back online again—

without any human intervention. Your users can now continue to shop on your site. In this chapter, we'll teach you how to build a high-availability architecture based on EC2 instances.

Virtual machines aren't highly available by default. The following scenarios could cause an outage of your virtual machine:

- A software issue causes the virtual machine's OS to fail.
- A software issue occurs on the host machine, causing the VM to crash (either the OS of the host machine crashes or the virtualization layer does).
- The computing, storage, or networking hardware of the physical host fails.
- Parts of the data center that the virtual machine depends on fail: network connectivity, the power supply, or the cooling system.

For example, if the computing hardware of a physical host fails, all EC2 instances running on this host will fail. If you're running an application on an affected virtual machine, this application will fail and experience downtime until somebody—probably you—intervenes by starting a new virtual machine on another physical host. To avoid downtimes, you should enable auto recovery or use multiple virtual machines.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

High availability describes a system that is operating with almost no downtime. Even if a failure occurs, the system can provide its services most of the time (for example, 99.99% over a year). Although a short interruption might be necessary to recover from a failure, there is no need for human interaction. The Harvard Research Group (HRG) defines high availability with the classification AEC-2, which requires an uptime of 99.99% over a year, or not more than 52 minutes and 35.7 seconds of downtime per year. You can achieve 99.99% uptime with EC2 instances if you follow the instructions in the rest of this chapter.

High availability vs. fault tolerance

A highly available system can recover from a failure automatically with a short downtime. A fault-tolerant system, in contrast, requires the system to provide its services without interruption in case of a component failure. We'll show you how to build a fault-tolerant system in chapter 16.

AWS offers tools for building highly available systems based on EC2 instances:

- Monitoring the health of virtual machines with CloudWatch and triggering recovery automatically if needed
- Building a highly available infrastructure by using groups of isolated data centers, called availability zones, within a region
- Using auto-scaling to guarantee a certain number of virtual machines will keep running, and replace failed instances automatically

14.1 Recovering from EC2 instance failure with CloudWatch

The EC2 service checks the status of every virtual machine automatically. System status checks are performed every minute and the results are available as CloudWatch metrics.

> **AWS CloudWatch**
>
> *AWS CloudWatch* is a service offering metrics, events, logs, and alarms for AWS resources. You used CloudWatch to monitor a Lambda function in chapter 7, and gained some insight into the current load of a relational database instance in chapter 11.

A *system status check* detects a loss of network connectivity or power, as well as software or hardware issues on the physical host. AWS then needs to repair failures detected by the system status check. One possible strategy to resolve such failures is to move the virtual machine to another physical host.

Figure 14.1 shows the process in the case of an outage affecting a virtual machine:

1. The physical hardware fails and causes the EC2 instance to fail as well.
2. The EC2 service detects the outage and reports the failure to a CloudWatch metric.
3. A CloudWatch alarm triggers recovery of the virtual machine.
4. The EC2 instance is launched on another physical host.
5. The EBS volume and Elastic IP stay the same, and are linked to the new EC2 instance.

After the recovery, a new EC2 instance is running with the same ID and private IP address. Data on network-attached EBS volumes is available as well. No data is lost because the EBS volume stays the same. EC2 instances with local disks (instance storage) aren't supported for this process. If the old EC2 instance was connected to an Elastic IP address, the new EC2 instance is connected to the same Elastic IP address.

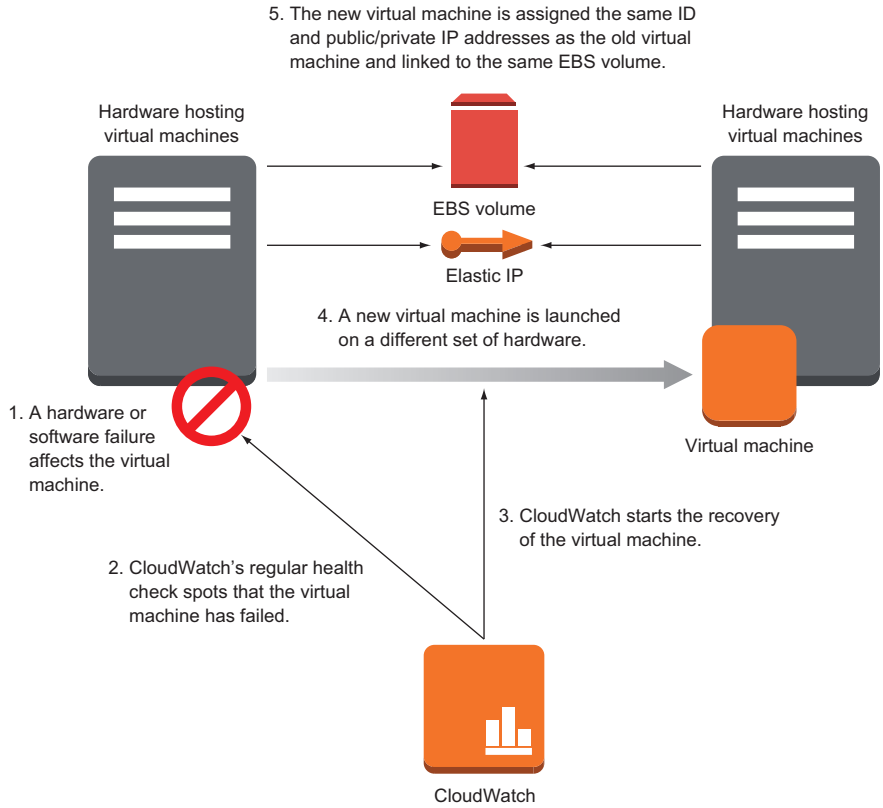


Figure 14.1 In the case of a hardware failure, CloudWatch triggers the recovery of the EC2 instance.

Requirements for recovering EC2 instances

An EC2 instance must meet the following requirements if you want to use the recovery feature:

- It must be running in a VPC network.
- The instance family must be C3, C4, C5, M3, M4, M5, R3, R4, T2, or X1. Other instance families aren't supported.
- The EC2 instance must use EBS volumes exclusively, because data on instance storage would be lost after the instance was recovered.

14.1.1 Creating a CloudWatch alarm to trigger recovery when status checks fail

A CloudWatch alarm consists of the following:

- A metric that monitors data (health check, CPU usage, and so on)
- A rule defining a threshold based on a statistical function over a period of time
- Actions to trigger if the state of the alarm changes (such as triggering a recovery of an EC2 instance if the state changes to `ALARM`)

The following states are available for an alarm:

- `OK`—Everything is fine; the threshold hasn't been reached.
- `INSUFFICIENT_DATA`—There isn't enough data to evaluate the alarm.
- `ALARM`—Something is broken: the threshold has been overstepped.

To monitor a VM's health and recover it in case the underlying host system fails, you can use a CloudWatch alarm like the one shown in listing 14.1. This listing is an excerpt from a CloudFormation template.

Listing 14.1 creates a CloudWatch alarm based on a metric called `StatusCheckFailed_System` (linked by attribute `MetricName`). This metric contains the results of the system status checks performed by the EC2 service every minute. If the check fails, a measurement point with value `1` is added to the metric `StatusCheckFailed_System`. Because the EC2 service publishes this metric, the `Namespace` is called `AWS/EC2` and the `Dimension` of the metric is the ID of a virtual machine.

The CloudWatch alarm checks the metric every 60 seconds, as defined by the `Period` attribute. As defined in `EvaluationPeriods`, the alarm will check the last five periods, the last 5 minutes in this example. The check runs a statistical function specified in `Statistic` on the time periods. The result of the statistical function, a minimum function in this case, is compared against `Threshold` using the chosen `ComparisonOperator`. If the result is negative, the alarm actions defined in `AlarmActions` are executed: in this case, the recovery of the virtual machine—a built-in action for EC2 instances.

Listing 14.1 Creating a CloudWatch alarm to monitor the health of an EC2 instance

```
# [...]
RecoveryAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    AlarmDescription: 'Recover EC2 instance ...'
    Namespace: 'AWS/EC2'
    MetricName: 'StatusCheckFailed_System'
    Statistic: Maximum
    Period: 60
    EvaluationPeriods: 5
```

Creates a CloudWatch alarm to monitor the VM's health

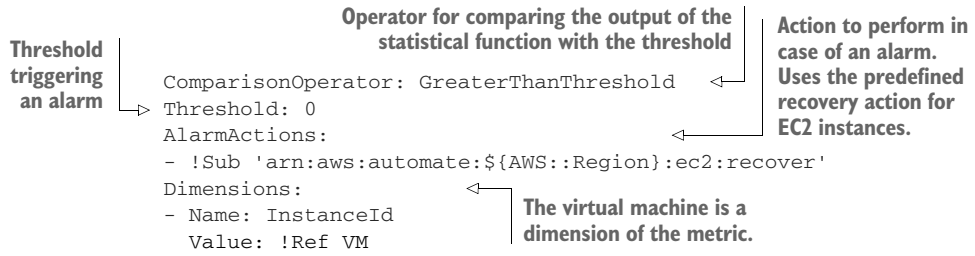
The metric to monitor is provided by the EC2 service with namespace AWS/EC2.

The name of the metric.

Statistical function to apply to the metric.

Number of time periods over which data is compared to the threshold

Duration the statistical function is applied, in seconds. Must be a multiple of 60.



In summary, AWS checks the status of the virtual machine every minute. The result of these checks is written to the `StatusCheckFailed_System` metric. The alarm checks this metric. If there are five consecutive failed checks, the alarm trips.

14.1.2 Monitoring and recovering a virtual machine based on a CloudWatch alarm

Suppose that your team is using an agile development process. To accelerate the process, your team decides to automate the testing, building, and deployment of the software. You've been asked to set up a continuous integration (CI) server. You've chosen to use Jenkins, an open source application written in Java that runs in a servlet container such as Apache Tomcat. Because you're using infrastructure as code, you're planning to deploy changes to your infrastructure with Jenkins as well.[1]

A Jenkins server is a typical use case for a high-availability setup. It's an important part of your infrastructure, because your colleagues won't be able to test and deploy new software if Jenkins suffers from downtime. But a short downtime in the case of a failure with automatic recovery won't hurt your business too much, so you don't need a fault-tolerant system. Jenkins is only an example. You can apply the same principles to any other applications where you can tolerate a short amount of downtime but still want to recover from hardware failures automatically. For example, we used the same approach for hosting FTP servers and VPN servers.

In this example, you'll do the following:

1. Create a virtual network in the cloud (VPC).
2. Launch a virtual machine in the VPC, and automatically install Jenkins during bootstrap.
3. Create a CloudWatch alarm to monitor the health of the virtual machine.

We'll guide you through these steps with the help of a CloudFormation template.

You can find the CloudFormation template for this example on GitHub and on S3. You can download a snapshot of the repository at http://mng.bz/x6RP. The file we're talking about is located at chapter14/recovery.yaml. On S3, the same file is located at http://mng.bz/994D.

[1] Learn more about Jenkins by reading its documentation at http://mng.bz/sVqd.

The following command creates a CloudFormation template which launches an EC2 instance with a CloudWatch alarm that triggers a recovery if the virtual machine fails. Replace *$Password* with a password consisting of 8–40 characters. The template automatically installs a Jenkins server while starting the virtual machine:

```
$ aws cloudformation create-stack --stack-name jenkins-recovery \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter14/recovery.yaml \
➥ --parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password
```

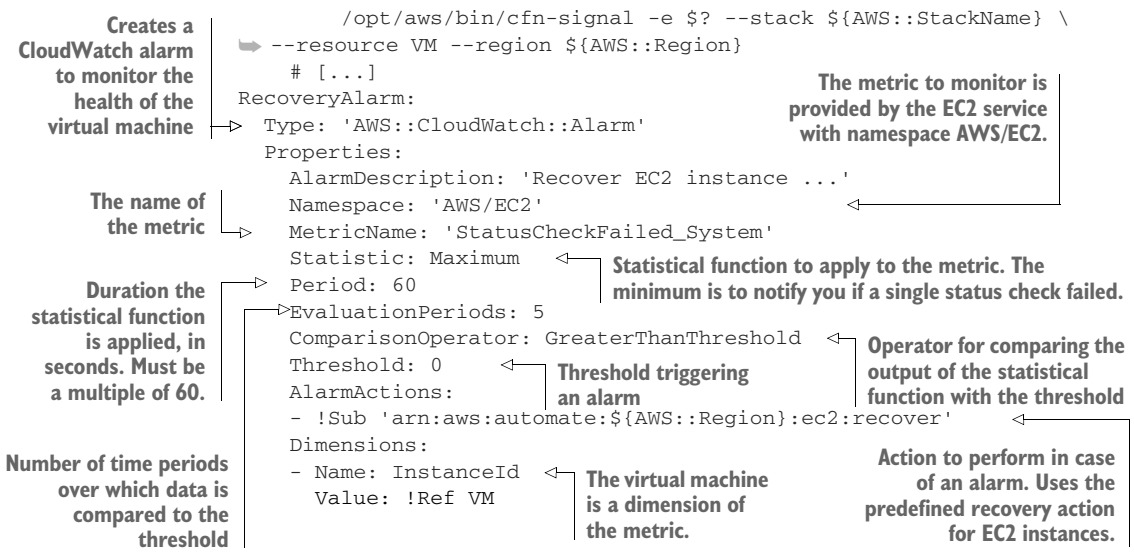
The CloudFormation template contains the definition of a private network and security configuration. But the most important parts of the template are these:

- A virtual machine with user data containing a Bash script, which installs a Jenkins server during bootstrapping
- A public IP address assigned to the EC2 instance, so you can access the new instance after a recovery using the same public IP address as before
- A CloudWatch alarm based on the system-status metric published by the EC2 service

The following listing shows the important parts of the template.

Listing 14.2 Starting an EC2 instance running a Jenkins CI server with a recovery alarm

```
#[...]
ElasticIP:                                  # The public IP address stays the same after recovery when using ElasticIP.
  Type: 'AWS::EC2::EIP'
  Properties:
    InstanceId: !Ref VM
    Domain: vpc
  DependsOn: GatewayToInternet
VM:                                         # Launches a virtual machine to run a Jenkins server
  Type: 'AWS::EC2::Instance'
  Properties:
    ImageId: 'ami-6057e21a'                 # Selects the AMI (in this case Amazon Linux)
    InstanceType: 't2.micro'                # Recovery is supported for t2 instance types.
    KeyName: mykey
    NetworkInterfaces:
    - AssociatePublicIpAddress: true
      DeleteOnTermination: true
      DeviceIndex: 0
      GroupSet:
      - !Ref SecurityGroup
      SubnetId: !Ref Subnet
    UserData:                               # User data containing a shell script that is executed during bootstrapping to install a Jenkins server
      'Fn::Base64': !Sub |
        #!/bin/bash -x
        bash -ex << "TRY"                   # Downloads and installs Jenkins
          wget -q -T 60 https://.../jenkins-1.616-1.1.noarch.rpm
          rpm --install jenkins-1.616-1.1.noarch.rpm
          # configure Jenkins [...]
          service jenkins start             # Starts Jenkins
        TRY
```



It will take a few minutes for the CloudFormation stack to be created and Jenkins to be installed on the virtual machine. Run the following command to get the output of the stack. If the output is null, retry after a few more minutes:

```
$ aws cloudformation describe-stacks --stack-name jenkins-recovery \
➥ --query "Stacks[0].Outputs"
```

If the query returns output like the following, containing a URL, a user, and a password, the stack has been created and the Jenkins server is ready to use. If you want more information during stack creation, we recommend you use the CloudFormation Management Console at https://console.aws.amazon.com/cloudformation/. Open the URL in your browser, and log in to the Jenkins server with user `admin` and the password you've chosen:

```
[
  {
    "Description": "URL to access web interface of Jenkins server.",
    "OutputKey": "JenkinsURL",
    "OutputValue": "http://54.152.240.91:8080"
  },
  {
    "Description": "Administrator user for Jenkins.",
    "OutputKey": "User",
    "OutputValue": "admin"
  },
  {
    "Description": "Password for Jenkins administrator user.",
    "OutputKey": "Password",
    "OutputValue": "********"
  }
]
```

Open this URL in your browser to access the web interface of the Jenkins server.

Use this user name to log in to the Jenkins server.

Use this password to log in to the Jenkins server.

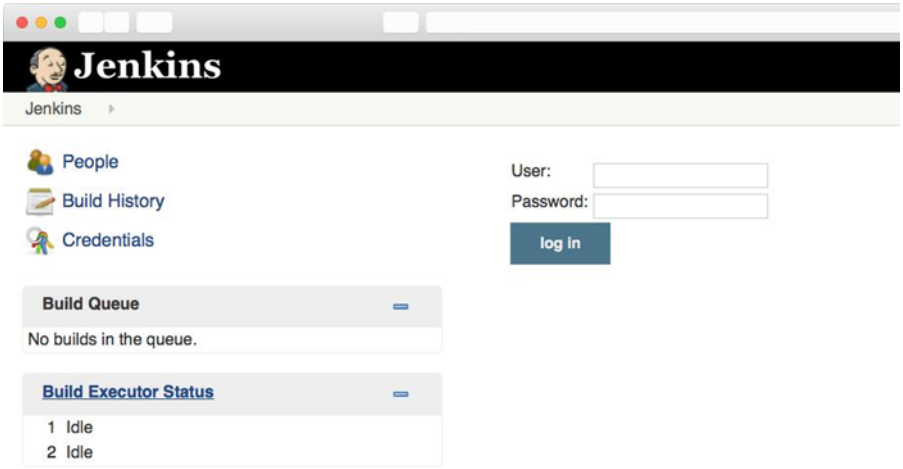


Figure 14.2 Web interface of the Jenkins server

You're now ready to create your first build job on the Jenkins server. To do so, you have to log in with the username and password from the previous output. Figure 14.2 shows the Jenkins server's login form.

Once you're logged in, you can create your first job by following these steps:

1. Click New Item in the navigation bar on the left.
2. Type `AWS in Action` as the name for the new job.
3. Select Freestyle Project as the job type, and click OK to save the job.

The Jenkins server runs on a virtual machine with automated recovery. If the virtual machine fails because of issues with the host system, it will be recovered with all data and the same public IP address. The URL doesn't change because you're using an Elastic IP for the virtual machine. All data is restored because the new virtual machine uses the same EBS volume as the previous virtual machine, so you can find your *AWS in Action* job again.

Unfortunately, you can't test the recovery process. The CloudWatch alarm monitors the health of the host system, which can only be controlled by AWS.

Cleaning up

Now that you've finished this example, it's time to clean up to avoid unwanted charges. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-recovery
$ aws cloudformation wait stack-delete-complete \
➥ --stack-name jenkins-recovery
```

Waits until the stack is deleted

14.2 Recovering from a data center outage

Recovering an EC2 instance after underlying software or hardware fails is possible using system status checks and CloudWatch, as described in the previous section. But what happens if the entire data center fails because of a power outage, a fire, or some other issue? Recovering a virtual machine as described in section 14.1 will fail because it tries to launch an EC2 instance in the same data center.

AWS is built for failure, even in the rare case that an entire data center fails. The AWS regions consist of multiple data centers grouped into availability zones. Auto-scaling helps you start virtual machines that can recover from a data center outage with only a short amount of downtime. There are two pitfalls when building a highly available setup over multiple availability zones:

1. Data stored on network-attached storage (EBS) won't be available after failing over to another availability zone by default. So you can end up having no access to your data (stored on EBS volumes) until the availability zone is back online (you won't lose your data in this case).
2. You can't start a new virtual machine in another availability zone with the same private IP address. As you've learned, subnets are bound to availability zones, and each subnet has a unique IP address range. By default, you can't keep the same public IP address automatically after a recovery, as was the case in the previous section with a CloudWatch alarm triggering a recovery.

In this section, you'll improve the Jenkins setup from the previous section, add the ability to recover from an outage of an entire availability zone, and work around the pitfalls afterward.

14.2.1 Availability zones: groups of isolated data centers

As you've learned, AWS operates multiple locations worldwide, called regions. You've used region US East (N. Virginia), also called us-east-1, if you've followed the examples so far. In total, there are 15 publicly available regions throughout North America, South America, Europe, and Asia Pacific.

Each region consists of multiple availability zones (AZs). You can think of an AZ as an isolated group of data centers, and a region as an area where multiple availability zones are located at a sufficient distance. The region us-east-1 consists of six availability zones (us-east-1a to us-east-1f) for example. The availability zone us-east-1a could be one data center, or many. We don't know because AWS doesn't make information about their data centers publicly available. So from an AWS user's perspective, you only know about regions and AZs.

The AZs are connected through low-latency links, so requests between different availability zones aren't as expensive as requests across the internet in terms of latency. The latency within an availability zone (such as from an EC2 instance to another EC2 instance in the same subnet) is lower compared to latency across AZs. The number of availability zones depends on the region. Most regions come with three or more

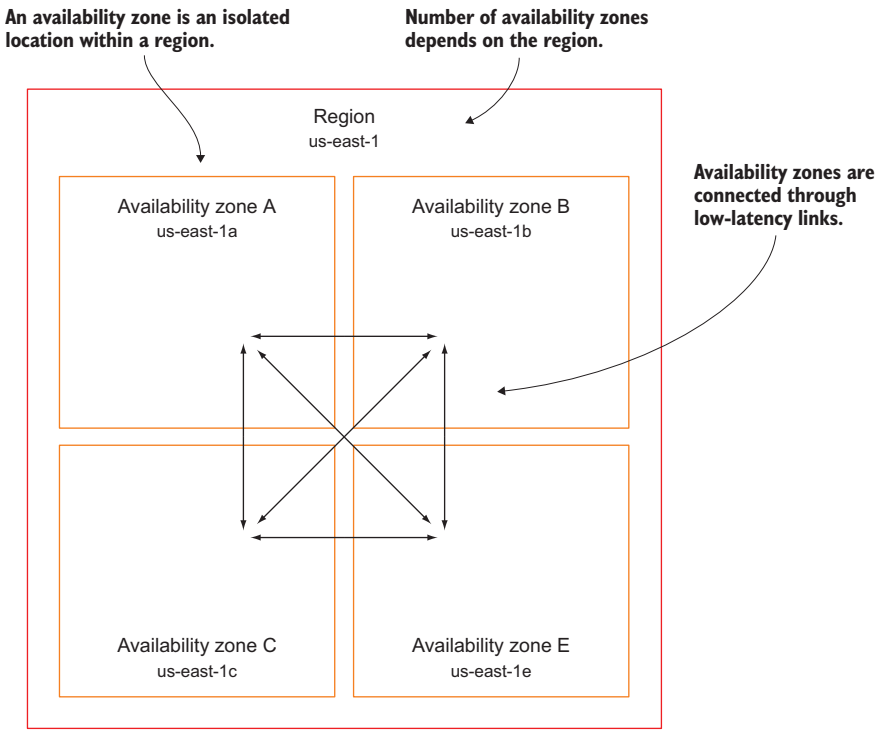


Figure 14.3 A region consists of multiple availability zones connected through low-latency links.

availability zones. When choosing a region, keep in mind that AWS also has regions with only two availability zones. This could become an issue if you want to run a distributed system that relies on consensus decisions. Figure 14.3 illustrates the concept of availability zones within a region.

Some AWS services are highly available or even fault-tolerant by default. Other services provide building blocks to achieve a highly available architecture. You can use multiple availability zones or even multiple regions to build a highly available architecture, as figure 14.4 shows:

- Some services operate globally over multiple regions: Route 53 (DNS) and CloudFront (CDN).
- Some services use multiple availability zones within a region so they can recover from an availability zone outage: S3 (object store) and DynamoDB (NoSQL database).
- The Relational Database Service (RDS) offers the ability to deploy a master-standby setup, called *Multi-AZ deployment*, so you can fail over into another availability zone with a short downtime if necessary.
- A virtual machine runs in a single availability zone. But AWS offers tools to build an architecture based on EC2 instances that can fail over into another availability zone.

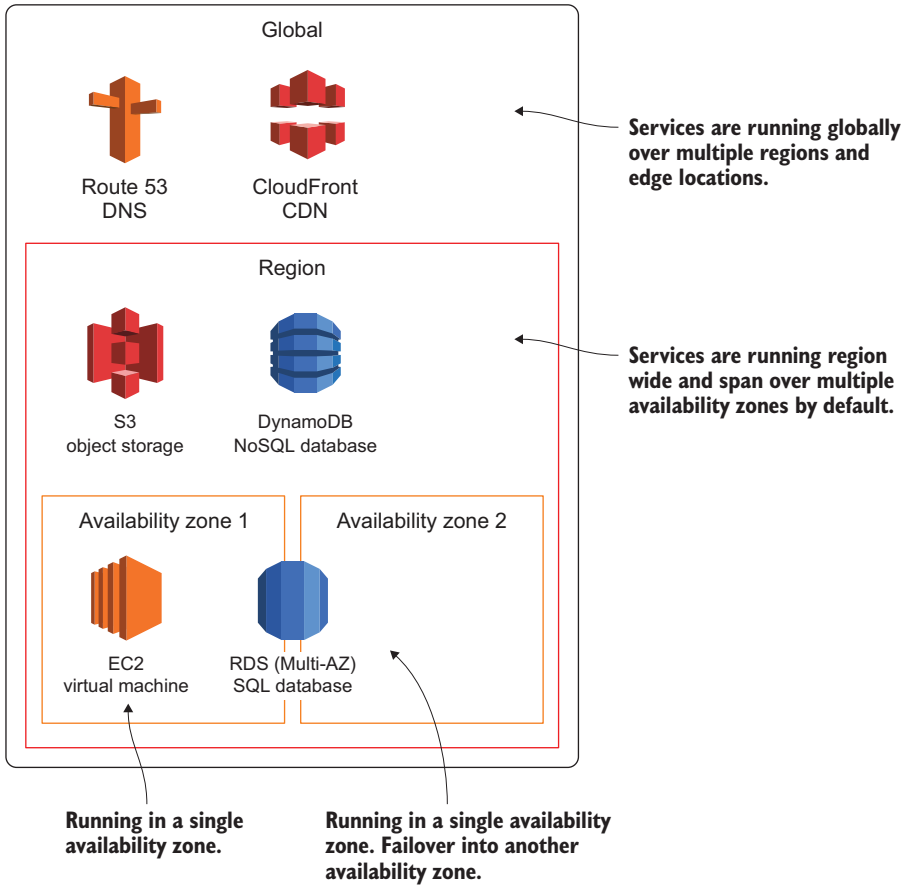


Figure 14.4 AWS services can operate in a single availability zone, over multiple availability zones within a region, or even globally.

The identifier for an availability zone consists of the identifier for the region (such as us-east-1) and a character (a, b, c, ...). So us-east-1a is the identifier for an availability zone in region us-east-1. To distribute resources across the different availability zones, the AZ identifier is generated randomly for each AWS account. This means us-east-1a points to a different availability zone in your AWS account than it does in our AWS account.

You can use the following commands to discover all regions available for your AWS account:

```
$ aws ec2 describe-regions
{
    "Regions": [
        {
            "Endpoint": "ec2.ap-south-1.amazonaws.com",
            "RegionName": "ap-south-1"
```

```
        },
        {
            "Endpoint": "ec2.eu-west-2.amazonaws.com",
            "RegionName": "eu-west-2"
        },
        {
            "Endpoint": "ec2.eu-west-1.amazonaws.com",
            "RegionName": "eu-west-1"
        },
        [...]
        {
            "Endpoint": "ec2.us-west-2.amazonaws.com",
            "RegionName": "us-west-2"
        }
    ]
}
```

To list all availability zones for a region, execute the following command and replace *$Region* with `RegionName` from the previous command:

```
$ aws ec2 describe-availability-zones --region $Region
{
    "AvailabilityZones": [
        {
            "State": "available",
            "ZoneName": "us-east-1a",
            "Messages": [],
            "RegionName": "us-east-1"
        },
        {
            "State": "available",
            "ZoneName": "us-east-1b",
            "Messages": [],
            "RegionName": "us-east-1"
        },
        [...]
        {
            "State": "available",
            "ZoneName": "us-east-1f",
            "Messages": [],
            "RegionName": "us-east-1"
        }
    ]
}
```

Before you start to create a high-availability architecture based on EC2 instances with failover to multiple availability zones, there is one more lesson to learn. If you define a private network in AWS with the help of the VPC service, you need to know the following:

- A VPC is always bound to a region.
- A subnet within a VPC is linked to an availability zone.
- A virtual machine is launched into a single subnet.

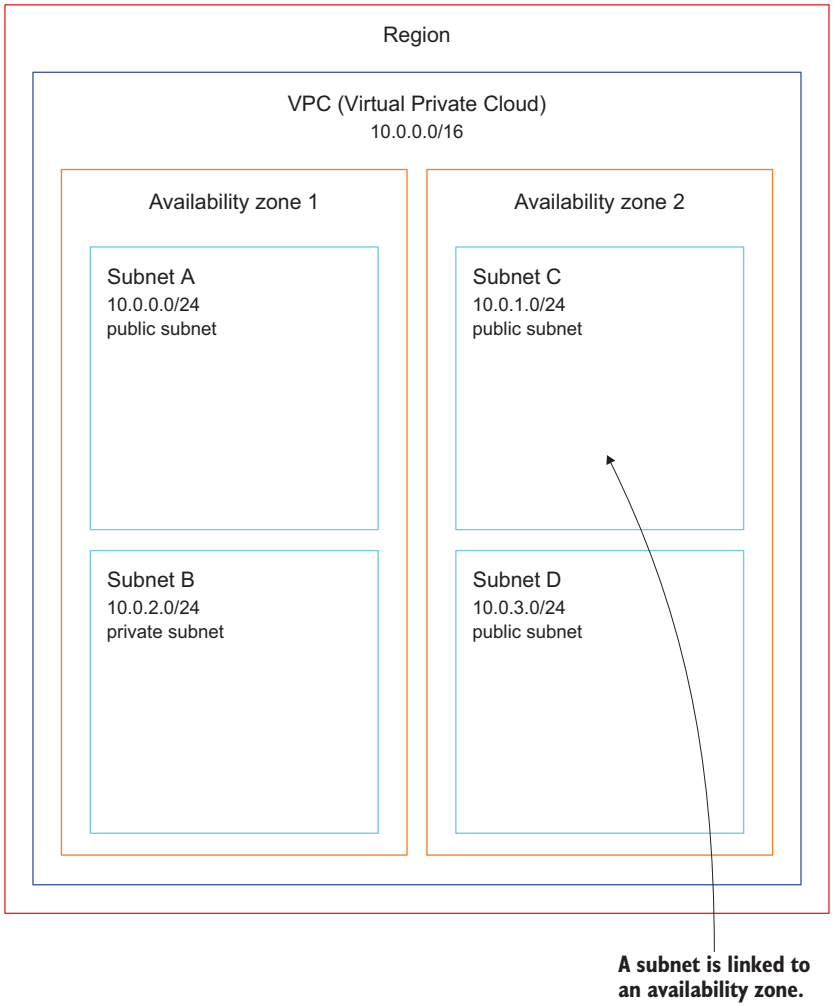


Figure 14.5 A VPC is bound to a region, and a subnet is linked to an availability zone.

Figure 14.5 illustrates these dependencies. Next, you'll learn how to launch a virtual machine that will automatically restart in another availability zone if a failure occurs.

14.2.2 Using auto-scaling to ensure that an EC2 instance is always running

Auto-scaling is part of the EC2 service and helps you to ensure that a specified number of EC2 instances is running even when availability zones become unavailable. You can use auto-scaling to launch a virtual machine and make sure a new instance is started if the original instance fails. You can use it to start virtual machines in multiple subnets. So in case of an outage of an entire availability zone, a new instance can be launched in another subnet in another availability zone.

To configure auto-scaling, you need to create two parts of the configuration:

- A *launch configuration* contains all information needed to launch an EC2 instance: instance type (size of virtual machine) and image (AMI) to start from.
- An *auto-scaling group* tells the EC2 service how many virtual machines should be started with a specific launch configuration, how to monitor the instances, and in which subnets EC2 instances should be started.

Figure 14.6 illustrates this process.

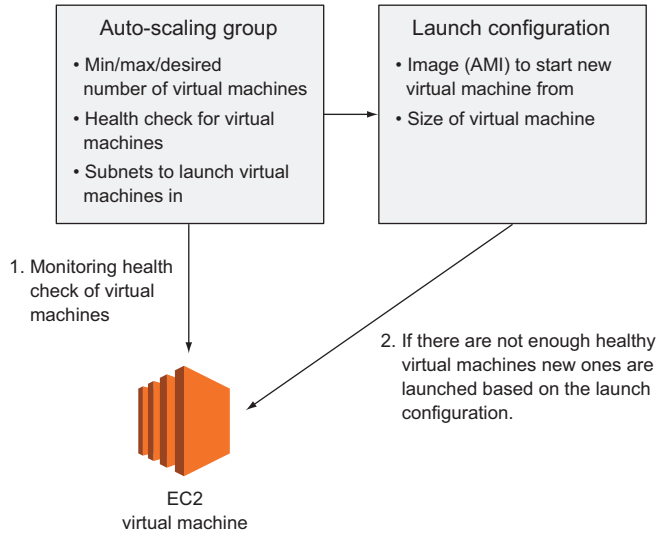


Figure 14.6 Auto-scaling ensures that a specified number of EC2 instances are running.

Listing 14.3 shows how to use auto-scaling to make sure a single EC2 instance is always running. The parameters are explained in table 14.1.

Table 14.1 Required parameters for the launch configuration and auto-scaling group

Context	Property	Description	Values
`LaunchConfiguration`	`ImageId`	The ID of the AMI the virtual machine should be started from.	Any AMI ID accessible from your account.
`LaunchConfiguration`	`InstanceType`	The size of the virtual machine.	All available instance sizes, such as t2.micro, m3.medium, and c3.large.
`AutoScalingGroup`	`DesiredCapacity`	The number of virtual machines that should run in the auto-scaling group at the moment.	Any positive integer. Use `1` if you want a single virtual machine to be started based on the launch configuration.

Table 14.1 Required parameters for the launch configuration and auto-scaling group *(continued)*

Context	Property	Description	Values
`AutoScalingGroup`	`MinSize`	The minimum value for the `DesiredCapacity`	Any positive integer. Use `1` if you want a single virtual machine to be started based on the launch configuration.
`AutoScalingGroup`	`MaxSize`	The maximum value for the `DesiredCapacity`	Any positive integer (greater than or equal to the `MinSize` value). Use 1 if you want a single virtual machine to be started based on the launch configuration.
`AutoScalingGroup`	`VPCZoneIdentifier`	The subnet IDs you want to start virtual machines in	Any subnet ID from a VPC from your account. Subnets must belong to the same VPC.
`AutoScalingGroup`	`HealthCheckType`	The health check used to identify failed virtual machines. If the health check fails, the auto-scaling group replaces the virtual machine with a new one.	`EC2` to use the status checks of the virtual machine, or `ELB` to use the health check of the load balancer (see chapter 16).

An auto-scaling group is also used if you need to scale the number of virtual machines based on usage of your system. You'll learn how to scale the number of EC2 instances based on current load in chapter 17. In this chapter, you only need to make sure a single virtual machine is always running. Because you need a single virtual machine, set the following parameters for auto-scaling to `1`:

- `DesiredCapacity`
- `MinSize`
- `MaxSize`

Listing 14.3 Configuring an auto-scaling group and a launch configuration

```
# [...]
LaunchConfiguration:
  Type: 'AWS::AutoScaling::LaunchConfiguration'
  Properties:
    ImageId: 'ami-6057e21a'
    InstanceType: 't2.micro'
    # [...]
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchConfigurationName: !Ref LaunchConfiguration
```

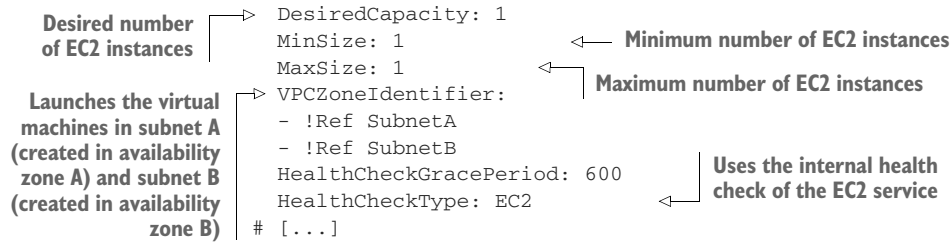
Select the AMI (in this case Amazon Linux).

Launch configuration used for auto-scaling.

Size of the virtual machine

Auto-scaling group responsible for launching the virtual machine

Link to the launch configuration.



The next section will reuse the Jenkins example from the beginning of the chapter to show you how high availability can be achieved with auto-scaling in practice.

14.2.3 Recovering a failed virtual machine to another availability zone with the help of auto-scaling

In the first part of the chapter, you used a CloudWatch alarm to trigger the recovery of a virtual machine that was running a Jenkins CI server, in case of a failure. This mechanism launches an identical copy of the original virtual machine if necessary. This is only possible in the same availability zone, because the private IP address and the EBS volume of a virtual machine are bound to a single subnet and a single availability zone. But suppose your team isn't happy about the fact that they won't be able to use the Jenkins server to test, build, and deploy new software in case of a unlikely availability zone outage. You begin looking for a tool that will let you recover in another availability zone.

Failing over into another availability zone is possible with the help of auto-scaling. You can find the CloudFormation template for this example on GitHub and on S3. You can download a snapshot of the repository at http://mng.bz/x6RP. The file we're talking about is located at chapter14/multiaz.yaml. On S3, the same file is located at http://mng.bz/994D.

Execute the following command to create a virtual machine that can recover in another availability zone if necessary. Replace *$Password* with a password consisting of 8–40 characters. The command uses the CloudFormation template shown in listing 14.4 to set up the environment.

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter14/multiaz.yaml \
➥ --parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password
```

You'll find both a launch configuration and an auto-scaling group in the CloudFormation template shown in listing 14.4. You already used the most important parameters for the launch configuration when starting a single virtual machine with a CloudWatch recovery alarm in the previous section:

- `ImageId`—ID of the image (AMI) for virtual machine
- `InstanceType`—Size of the virtual machine
- `KeyName`—Name of the SSH key pair
- `SecurityGroupIds`—Link to the security groups
- `UserData`—Script executed during bootstrap to install the Jenkins CI server

There is one important difference between the definition of a single EC2 instance and the launch configuration: the subnet for the virtual machine isn't defined in the launch configuration, but rather in the auto-scaling group.

Listing 14.4 Launching a Jenkins VM with auto-scaling in two AZs

```
# [...]
LaunchConfiguration:
  Type: 'AWS::AutoScaling::LaunchConfiguration'
  Properties:
    InstanceMonitoring: false
    ImageId: 'ami-6057e21a'
    KeyName: mykey
    SecurityGroups:
    - !Ref SecurityGroup
    AssociatePublicIpAddress: true
    InstanceType: 't2.micro'
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -x
        bash -ex << "TRY"
          wget -q -T 60 https://.../jenkins-1.616-1.1.noarch.rpm
          rpm --install jenkins-1.616-1.1.noarch.rpm
          # [...]
          service jenkins start
        TRY
        /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
➥ --resource AutoScalingGroup --region ${AWS::Region}
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    LaunchConfigurationName: !Ref LaunchConfiguration
    Tags:
    - Key: Name
      Value: 'jenkins-multiaz'
      PropagateAtLaunch: true
    DesiredCapacity: 1
    MinSize: 1
    MaxSize: 1
    VPCZoneIdentifier:
    - !Ref SubnetA
    - !Ref SubnetB
    HealthCheckGracePeriod: 600
    HealthCheckType: EC2
  CreationPolicy:
    ResourceSignal:
      Timeout: PT10M
```

Launch configuration used for auto-scaling.

Select the AMI (in this case Amazon Linux).

By default, EC2 sends metrics to CloudWatch every 5 minutes. But you can enable detailed instance monitoring to get metrics every minute for an additional cost.

Key for the SSH connections to the virtual machine

Enables the public IP address for the virtual machine

Security groups attached to the virtual machine

Size of the virtual machine

User data containing a script executed during bootstrapping that installs a Jenkins server on the virtual machine

Installs Jenkins

Link to the launch configuration

Auto-scaling group responsible for launching the virtual machine

Tags for the auto-scaling group

Attaches the same tags to the virtual machine started by this auto-scaling group

Desired number of EC2 instances

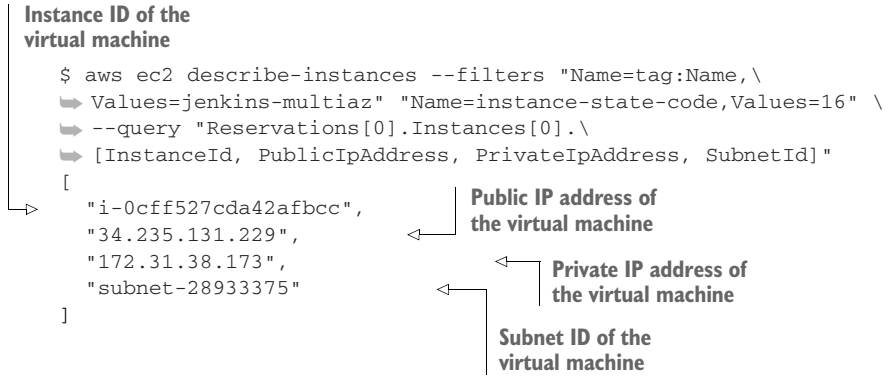
Minimum number of EC2 instances

Maximum number of EC2

Launches the virtual machines in subnet A (created in availability zone A) and subnet B (created in availability zone B)

Uses the internal health check of the EC2 service to discover issues with the virtual machine

The creation of the CloudFormation stack will take a few minutes—time to grab some coffee or tea and take a short break. Execute the following command to grab the public IP address of the virtual machine. If no IP address appears, the virtual machine isn't started yet. Wait another minute, and try again:




```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0cff527cda42afbcc",
  "34.235.131.229",
  "172.31.38.173",
  "subnet-28933375"
]
```

Open http://*$PublicIP*:8080 in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface for the Jenkins server appears.

Execute the following command to terminate the virtual machine and test the recovery process with auto-scaling. Replace `$InstanceId` with the instance ID from the output of the previous `describe` command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

After a few minutes, the auto-scaling group detects that the virtual machine was terminated and starts a new virtual machine. Rerun the `describe-instances` command until the output contains a new running virtual machine:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0293522fad287bdd4",
  "52.3.222.162",
  "172.31.37.78",
  "subnet-45b8c921"
]
```

The instance ID, the public IP address, the private IP address, and probably even the subnet ID have changed for the new instance. Open http://*$PublicIP*:8080 in your browser, and replace `$PublicIP` with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears.

You've now built a highly available architecture consisting of an EC2 instance with the help of auto-scaling. There are two issues with the current setup:

- *The Jenkins server stores data on disk.* When a new virtual machine is started to recover from a failure, this data is lost because a new disk is created.
- *The public and private IP addresses of the Jenkins server change after a new virtual machine is started for recovery.* The Jenkins server is no longer available under the same endpoint.

You'll learn how to solve these problems in the next part of the chapter.

Cleaning up

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz
$ aws cloudformation wait stack-delete-complete \
➥ --stack-name jenkins-multiaz
```

Waits until the stack is deleted

14.2.4 Pitfall: recovering network-attached storage

The EBS service offers network-attached storage for virtual machines. Remember that EC2 instances are linked to a subnet, and the subnet is linked to an availability zone. EBS volumes are also located in a single availability zone only. If your virtual machine is started in another availability zone because of an outage, the EBS volume cannot be accessed from the other availability zone. Let's say your Jenkins data is stored on an EBS volume in availability zone us-east-1a. As long as you have an EC2 instance running in the same availability zone, you can attach the EBS volume. But if this availability zone becomes unavailable, and you start a new EC2 instance in availability zone us-east-1b, you can't access that EBS volume in us-east-1a, which means that you can't recover Jenkins because you don't have access to the data. See figure 14.7.

Don't mix availability and durability guarantees

An EBS volume is guaranteed to be available for 99.999% of the time. So in case of an availability zone outage, the volume is no longer available. This does not imply that you lose any data. As soon as the availability zone is back online, you can access the EBS volume again with all its data.

An EBS volume guarantees that you won't lose any data in 99.9% of the time. This guarantee is called the durability of the EBS volume. If you have 1,000 volumes in use, you can expect that you will lose one of the volumes and its data a year.

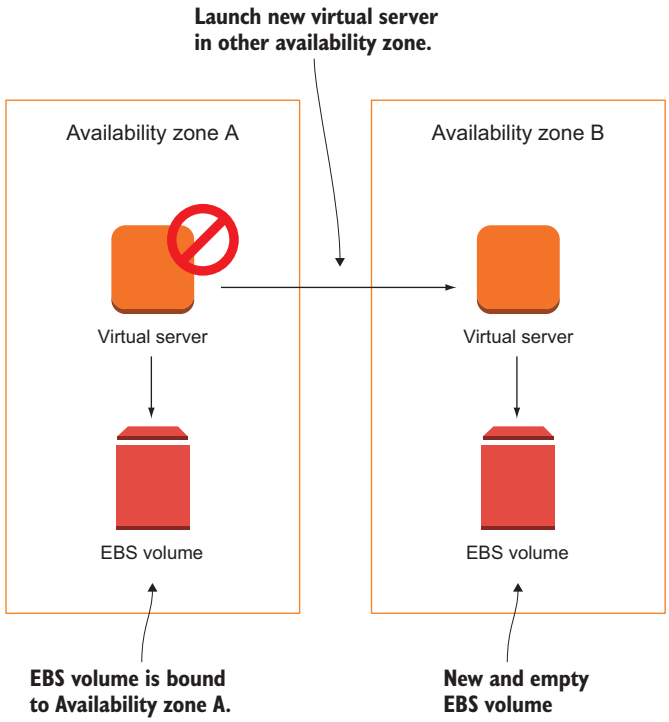


Figure 14.7 An EBS volume is only available in a single availability zone.

There are multiple solutions for this problem:

1. Outsource the state of your virtual machine to a managed service that uses multiple availability zones by default: RDS, DynamoDB (NoSQL database), EFS (NFSv4.1 share), or S3 (object store).
2. Create snapshots of your EBS volumes regularly, and use these snapshots if an EC2 instance needs to recover in another availability zone. EBS snapshots are stored on S3, thus available in multiple availability zones. If the EBS volume is the root volume of the ECS instance, create AMIs to back up the EBS volume instead of a snapshot.
3. Use a distributed third-party storage solution to store your data in multiple availability zones: GlusterFS, DRBD, MongoDB, and so on.

The Jenkins server stores data directly on disk. To outsource the state of the virtual machine, you can't use RDS, DynamoDB, or S3; you need a block-level storage solution instead. As you've learned, an EBS volume is only available in a single availability zone, so this isn't the best fit for the problem. But do you remember EFS from chapter 10? EFS provides block-level storage (over NFSv4.1) and replicates your data automatically between availability zones in a region.

AWS is a fast-growing platform

When we wrote the first edition of this book, EFS was not available. There was basically no easy way to share a filesystem between multiple EC2 instances. As you can imagine, many customers complained about this to AWS. Amazon is proud to be customer obsessed, which means that they listen to their customers carefully. If enough customers need a solution, AWS will deliver a solution. That's why you should follow the new features that are released daily. A problem that was hard to solve yesterday might now be solved by AWS natively. The best place to get updates about AWS is the AWS blog at https://aws.amazon.com/blogs/.

To embed EFS into the Jenkins setup, shown in listing 14.5, you have to make three modifications to the Multi-AZ template from the previous section:

1. Create an EFS filesystem.
2. Create EFS mount targets in each availability zone.
3. Adjust the user data to mount the EFS filesystem. Jenkins stores all its data under `/var/lib/jenkins`.

Listing 14.5 Store Jenkins state on EFS

```
# [...]
FileSystem:
  Type: 'AWS::EFS::FileSystem'
  Properties: {}
MountTargetSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
  Properties:
    GroupDescription: 'EFS Mount target'
    SecurityGroupIngress:
    - FromPort: 2049
      IpProtocol: tcp
      SourceSecurityGroupId: !Ref SecurityGroup
      ToPort: 2049
    VpcId: !Ref VPC
MountTargetA:
  Type: 'AWS::EFS::MountTarget'
  Properties:
    FileSystemId: !Ref FileSystem
    SecurityGroups:
    - !Ref MountTargetSecurityGroup
    SubnetId: !Ref SubnetA
MountTargetB:
  Type: 'AWS::EFS::MountTarget'
  Properties:
    FileSystemId: !Ref FileSystem
    SecurityGroups:
    - !Ref MountTargetSecurityGroup
    SubnetId: !Ref SubnetB
# [...]
```

Create the EFS filesystem.

The file system is protected by a security group.

Allows traffic only from the Jenkins EC2 Instances

A mount target is created in subnet A.

Mount target in subnet B

```
LaunchConfiguration:
Type: 'AWS::AutoScaling::LaunchConfiguration'
Properties:
  # [...]
  UserData:
    'Fn::Base64': !Sub |
      #!/bin/bash -x
      bash -ex << "TRY"
        wget -q -T 60 https://.../jenkins-1.616-1.1.noarch.rpm
        rpm --install jenkins-1.616-1.1.noarch.rpm
        while ! nc -z \
➥ ${FileSystem}.efs.${AWS::Region}.amazonaws.com 2049; \
➥ do sleep 10; done
        sleep 10
        echo -n "${FileSystem}.efs.${AWS::Region}.amazonaws.com:/ \
➥ /var/lib/jenkins" >> /etc/fstab
        echo " nfs4 nfsvers=4.1,rsize=1048576,wsize=1048576,hard, \
➥ timeo=600,retrans=2,_netdev 0 0" >> /etc/fstab
        mount -a
        chown jenkins:jenkins /var/lib/jenkins/
        # [...]
        service jenkins start
      TRY
      /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
➥ --resource AutoScalingGroup --region ${AWS::Region}
```

Waits until EFS file system is available

Installs Jenkins

Mounts EFS file system

Execute the following command to create the new Jenkins setup that stores state on EFS. Replace *$Password* with a password consisting of 8–40 characters.

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz-efs \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter14/multiaz-efs.yaml \
➥ --parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password
```

The creation of the CloudFormation stack will take a few minutes. Run the following command to get the public IP address of the virtual machine. If no IP address appears, the virtual machine isn't started yet. In this case, please wait another minute, and try again:

Instance ID of the virtual machine

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz-efs" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-0efcd2f01a3e3af1d",
  "34.236.255.218",
  "172.31.37.225",
  "subnet-0997e66d"
]
```

Public IP address of the virtual machine

Private IP address of the virtual machine

Subnet ID of the virtual machine

Open http://$PublicIP:8080 in your browser, and replace *$PublicIP* with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears.

Now, create a new Jenkins job by following these steps:

1. Open http://*$PublicIP:8080*/newJob in your browser, and replace *$PublicIP* with the public IP address from the output of the previous `describe` command.
2. Log in with user `admin` and the password you chose when starting the CloudFormation template.
3. Type in `AWS in Action` as the name for the new job.
4. Select Freestyle Project as the job type, and click OK to save the job.

You've made some changes to the state of Jenkins stored on EFS. Now, terminate the EC2 instance with the following command and you will see that Jenkins recovers from the failure without data loss. Replace *$InstanceId* with the instance ID from the output of the previous `describe` command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

After a few minutes, the auto-scaling group detects that the virtual machine was terminated and starts a new virtual machine. Rerun the `describe-instances` command until the output contains a new running virtual machine:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz-efs" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].\
➥ [InstanceId, PublicIpAddress, PrivateIpAddress, SubnetId]"
[
  "i-07ce0865adf50cccf",
  "34.200.225.247",
  "172.31.37.199",
  "subnet-0997e66d"
]
```

The instance ID, the public IP address, the private IP address, and probably even the subnet ID have changed for the new instance. Open http://*$PublicIP*:8080 in your browser, and replace *$PublicIP* with the public IP address from the output of the previous `describe-instances` command. The web interface from the Jenkins server appears and it still contains the `AWS in Action` job you created recently.

You've now built a highly available architecture consisting of an EC2 instance with the help of auto-scaling. State is now stored on EFS and is no longer lost when an EC2 instance is replaced. There is one issue left:

- The public and private IP addresses of the Jenkins server change after a new virtual machine is started for recovery. The Jenkins server is no longer available under the same endpoint.

Cleaning up

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz-efs
$ aws cloudformation wait stack-delete-complete \
➥ --stack-name jenkins-multiaz-efs
```

Waits until the stack is deleted

You'll learn how to solve the last issue next.

14.2.5 Pitfall: network interface recovery

Recovering a virtual machine using a CloudWatch alarm in the same availability zone, as described at the beginning of this chapter, is easy because the private IP address and the public IP address stay the same automatically. You can use these IP addresses as an endpoint to access the EC2 instance even after a failover.

You can't do this when using auto-scaling to recover from a EC2 instance or availability zone outage. If a virtual machine has to be started in another availability zone, it must be started in another subnet. So it's not possible to use the same private IP address for the new virtual machine, as figure 14.8 shows.

By default, you also can't use an Elastic IP as a public IP address for a virtual machine launched by auto-scaling. But the requirement for a static endpoint to receive requests is common. For the use case of a Jenkins server, developers want to bookmark an IP address or a hostname to reach the web interface. There are different possibilities for providing a static endpoint when using auto-scaling to build high availability for a single virtual machine:

- Allocate an Elastic IP, and associate this public IP address during the bootstrap of the virtual machine.
- Create or update a DNS entry linking to the current public or private IP address of the virtual machine.
- Use an Elastic Load Balancer (ELB) as a static endpoint that forwards requests to the current virtual machine.

To use the second solution, you need to link a domain with the Route 53 (DNS) service; we've chosen to skip this solution because you need a registered domain to implement it. The ELB solution is covered in chapter 15, so we'll skip it in this chapter as well. We'll focus on the first solution: allocating an Elastic IP and associating this public IP address during the virtual machine's bootstrap.

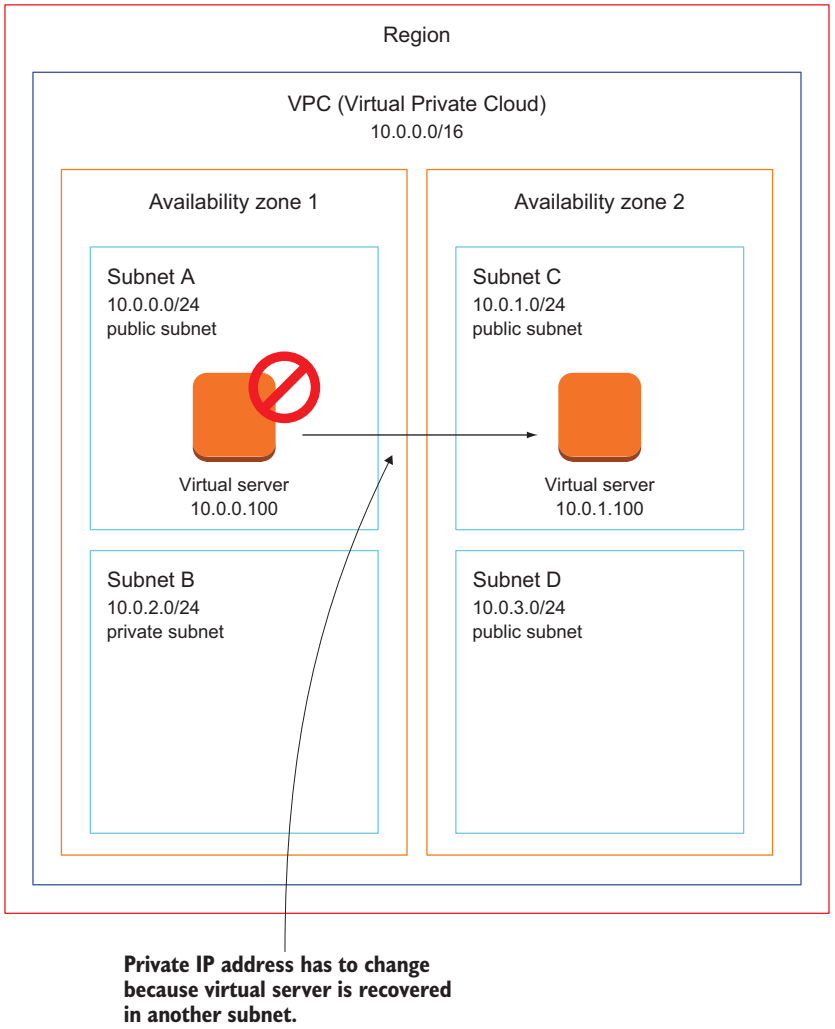


Figure 14.8 The virtual machine starts in another subnet in case of a failover and changes the private IP address.

Execute the following command to create the Jenkins setup based on auto-scaling again, using an Elastic IP address as static endpoint:

```
$ aws cloudformation create-stack --stack-name jenkins-multiaz-efs-eip \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter14/multiaz-efs-eip.yaml \
➥ --parameters ParameterKey=JenkinsAdminPassword,ParameterValue=$Password \
➥ --capabilities CAPABILITY_IAM
```

The command creates a stack based on the template shown in listing 14.6. The differences from the original template spinning up a Jenkins server with auto-scaling are as follows:

- Allocating an Elastic IP
- Adding the association of an Elastic IP to the script in the user data
- Creating an IAM role and policy to allow the EC2 instance to associate an Elastic IP

Listing 14.6 Using an EIP as a static endpoint for a VM launched by auto-scaling

```
# [...]
IamRole:                                    <-- Creates an IAM role used by the EC2 instance
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
      - Effect: Allow
        Principal:
          Service: 'ec2.amazonaws.com'
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: root
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Action: 'ec2:AssociateAddress'    <-- Associating an Elastic IP is allowed for EC2 instances using this IAM role.
          Resource: '*'
          Effect: Allow
IamInstanceProfile:
  Type: 'AWS::IAM::InstanceProfile'
  Properties:
    Roles:                                  <-- Allocates an Elastic IP for the virtual machine running Jenkins
    - !Ref IamRole
ElasticIP:
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: vpc                             <-- Creates an Elastic IP for VPC
LaunchConfiguration:
  Type: 'AWS::AutoScaling::LaunchConfiguration'
  Properties:
    InstanceMonitoring: false
    IamInstanceProfile: !Ref IamInstanceProfile
    ImageId: 'ami-6057e21a'
    KeyName: mykey
    SecurityGroups:
    - !Ref SecurityGroup
    AssociatePublicIpAddress: true
    InstanceType: 't2.micro'
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -x
```

```
      bash -ex << "TRY"
        INSTANCE_ID="$(curl -s http://169.254.169.254/\
➥ latest/meta-data/instance-id)"
        aws --region ${AWS::Region} ec2 associate-address \
➥ --instance-id $INSTANCE_ID \
➥ --allocation-id ${ElasticIP.AllocationId}
        # [...]
        service jenkins start
      TRY
      /opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} \
➥ --resource AutoScalingGroup --region ${AWS::Region}
```

Gets the instance ID from the instance metadata

Associates the Elastic IP with the virtual machine

If the query returns output as shown in the following listing, containing a URL, a user, and a password, the stack has been created and the Jenkins server is ready to use. Open the URL in your browser, and log in to the Jenkins server with user `admin` and the password you've chosen. If the output is `null`, try again in a few minutes:

```
$ aws cloudformation describe-stacks --stack-name jenkins-multiaz-efs-eip \
➥ --query "Stacks[0].Outputs"
```

You can now test whether the recovery of the virtual machine works as expected. To do so, you'll need to know the instance ID of the running virtual machine. Run the following command to get this information:

```
$ aws ec2 describe-instances --filters "Name=tag:Name,\
➥ Values=jenkins-multiaz-efs-eip" "Name=instance-state-code,Values=16" \
➥ --query "Reservations[0].Instances[0].InstanceId" --output text
```

Execute the following command to terminate the virtual machine and test the recovery process triggered by auto-scaling. Replace *`$InstanceId`* with the instance from the output of the previous command:

```
$ aws ec2 terminate-instances --instance-ids $InstanceId
```

Wait a few minutes for your virtual machine to recover. Because you're using an Elastic IP assigned to the new virtual machine on bootstrap, you can open the same URL in your browser as you did before the termination of the old instance.

Cleaning up

It's time to clean up to avoid unwanted costs. Execute the following command to delete all resources corresponding to the Jenkins setup:

```
$ aws cloudformation delete-stack --stack-name jenkins-multiaz-efs-eip
$ aws cloudformation wait stack-delete-complete \
➥ --stack-name jenkins-multiaz-efs-eip
```

Waits until the stack is deleted

Now the public IP address of your virtual machine running Jenkins won't change, even if the running virtual machine needs to be replaced by another virtual machine in another availability zone.

14.3 Analyzing disaster-recovery requirements

Before you begin implementing highly available or even fault-tolerant architectures on AWS, you should start by analyzing your disaster-recovery requirements. Disaster recovery is easier and cheaper in the cloud than in a traditional data center, but building for high availability increases the complexity and therefore the initial costs as well as the operating costs of your system. The recovery time objective (RTO) and recovery point objective (RPO) are standards for defining the importance of disaster recovery from a business point of view.

Recovery time objective (RTO) is the time it takes for a system to recover from a failure; it's the length of time until the system reaches a working state again, defined as the system service level, after an outage. In the example with a Jenkins server, the RTO would be the time until a new virtual machine is started and Jenkins is installed and running after a virtual machine or an entire availability zone goes down.

Recovery point objective (RPO) is the acceptable data-loss time caused by a failure. The amount of data loss is measured in time. If an outage happens at 10:00 a.m. and the system recovers with a data snapshot from 09:00 a.m., the time span of the data loss is one hour. In the example of a Jenkins server using auto-scaling, the RPO would be zero, because data is stored on EFS and is not lost during an AZ outage. Figure 14.9 illustrates the definitions of RTO and RPO.

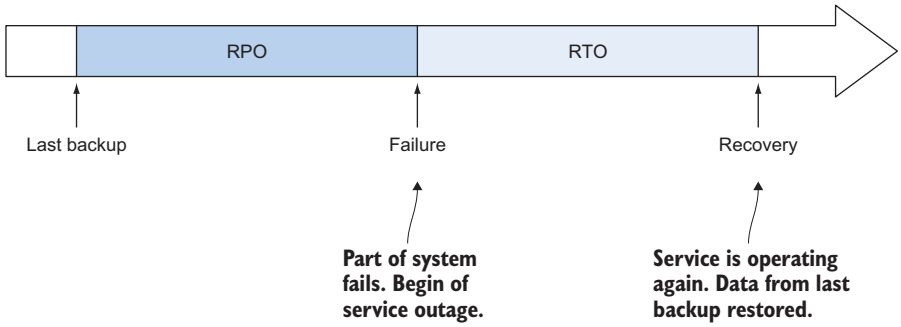


Figure 14.9 Definitions of RTO and RPO

14.3.1 RTO and RPO comparison for a single EC2 instance

You've learned about two possible solutions for making a single EC2 instance highly available. When choosing the solution, you have to know the application's business requirements. Can you tolerate the risk of being unavailable if an availability zone goes down? If so, EC2 instance recovery is the simplest solution where you don't lose any data. If your application needs to survive an unlikely availability zone outage, your safest bet is auto-scaling with data stored on EFS. But this also has performance

impacts compared to storing data on EBS volumes. As you can see, there is no one-size-fits-all solution. You have to pick the solution that fits your business problem best. Table 14.2 compares the solutions.

Table 14.2 Comparison of high availability for a single EC2 instance

	RTO	RPO	Availability
EC2 instance, data stored on EBS root volume: recovery triggered by a CloudWatch alarm	About 10 minutes	No data loss	Recovers from a failure of a virtual machine but not from an outage of an entire availability zone
EC2 instance, data stored on EBS root volume: recovery triggered by auto-scaling	About 10 minutes	All data is lost.	Recovers from a failure of a virtual machine and from an outage of an entire availability zone
EC2 instance, data stored on EBS root volume with regular snapshots: recovery triggered by auto-scaling	About 10 minutes	Realistic time span for snapshots is between 30 minutes and 24 hours.	Recovers from a failure of a virtual machine and from an outage of an entire availability zone
EC2 instance, data stored on EFS filesystem: recovery triggered by auto-scaling	About 10 minutes	No data loss.	Recovers from a failure of a virtual machine and from an outage of an entire availability zone

If you want to be able to recover from an outage of an availability zone and need to decrease the RPO, you should try to achieve a stateless server. Using storage services like RDS, EFS, S3, and DynamoDB can help you to do so. See part 3 if you need help with using these services.

Summary

- A virtual machine fails if the underlying hardware or software fails.
- You can recover a failed virtual machine with the help of a CloudWatch alarm: By default, data stored on EBS, as well as the private and public IP addresses, stays the same.
- An AWS region consists of multiple isolated groups of data centers called availability zones.
- Recovering from an availability zone outage is possible when using multiple availability zones.
- Some AWS services use multiple availability zones by default, but virtual machines run in a single availability zone.
- You can use auto-scaling to guarantee that a single virtual machine is always running even if an availability zone fails. The pitfalls are that you can no longer blindly rely on EBS volumes and by default, IP addresses will change.
- Recovering data in another availability zone is tricky when stored on EBS volumes instead of managed storage services like RDS, EFS, S3, and DynamoDB.

15 Decoupling your infrastructure: Elastic Load Balancing and Simple Queue Service

This chapter covers

- The reasons for decoupling a system
- Synchronous decoupling with load balancers to distribute requests
- Hiding your backend from users and message producers
- Asynchronous decoupling with message queues to buffer message peaks

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

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

The problem with our meeting is that it's tightly coupled to a location. We live in Germany; you probably don't. We can solve that issue by decoupling our meeting from the location. So we change plans and schedule a Google Hangout session. Now we must:

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

Google Hangouts (and other video/voice chat services) does synchronous decoupling. It removes the need to be at the same place, while still requiring us to meet at the same time.

We can even decouple from time by using email. Now we must:

- Find each other via email

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

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

NOTE To fully understand this chapter, you'll need to have read and understood the concept of auto-scaling covered in chapter 14.

So far you learned about two ways to decouple a meeting:

- *No decoupling*—We have to be at the same place (the cafe), at the same time (3 p.m.), and find each other (I have black hair and I'm wearing a white shirt).
- *Synchronous decoupling*—We can now be at different places. But still, we have to find a common time (3 p.m.), and find each other (exchange Skype IDs).
- *Asynchronous decoupling*—We can be at different places and now also don't have to find a common time. We only have to find each other (exchange email addresses).

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

- A public IP address is like the location of our meeting: To make a request to a web server, you must know its public IP address, and the virtual machine must be connected to that address. If you want to change the public IP address, both parties are involved in making the appropriate changes.

- If you want to make a request to a web server, the web server must be online at the same time. Otherwise your request will be rejected. There are many reasons why a web server can be offline: someone might be installing updates, a hardware failure, and so on.

AWS offers a solution for synchronous and asynchronous decoupling. The *Elastic Load Balancing* (ELB) service provides different types of load balancers that sit between your EC2 instances and the client to decouple your requests synchronously. For asynchronous decoupling, AWS offers a *Simple Queue Service* (SQS) that provides a message queue infrastructure. You'll learn about both services in this chapter. Let's start with ELB.

15.1 Synchronous decoupling with load balancers

Exposing a single EC2 instance running a web server to the outside world introduces a dependency: your users now depend on the public IP address of the EC2 instance. As soon as you distribute the public IP address to your users, you can't change it anymore. You're faced with the following issues:

- Changing the public IP address is no longer possible because many clients rely on it.
- If you add an additional EC2 instance (and IP address) to handle the increasing load, it's ignored by all current clients: they're still sending all requests to the public IP address of the first server.

You can solve these issues with a DNS name that points to your server. But DNS isn't fully under your control. DNS resolvers cache responses. DNS servers cache entries, and sometimes they don't respect your time to live (TTL) settings. For example, you might ask DNS servers to only cache the name-to-IP address mapping for one minute, but some DNS servers might use a minimum cache of one day. A better solution is to use a load balancer.

A load balancer can help decouple a system where the requester awaits an immediate response. Instead of exposing your EC2 instances (running web servers) to the outside world, you only expose the load balancer to the outside world. The load balancer then forwards requests to the EC2 instances behind it. Figure 15.1 shows how this works.

The requester (such as a web browser) send an HTTP request to the load balancer. The load balancer then selects one of the EC2 instances and copies the original HTTP request to send to the EC2 instance that it selected. The EC2 instance then processes the request and sends a response. The load balancer receives the response, and sends the same response to the original requester.

AWS offers different types of load balancers through the Elastic Load Balancing (ELB) service. All load balancer types are fault-tolerant and scalable. They differ mainly in the protocols they support:

- *Application Load Balancer (ALB)—HTTP, HTTPS*
- *Network Load Balancer (NLB)*—TCP
- *Classic Load Balancer (CLB)*—HTTP, HTTPS, TCP, TCP+TLS

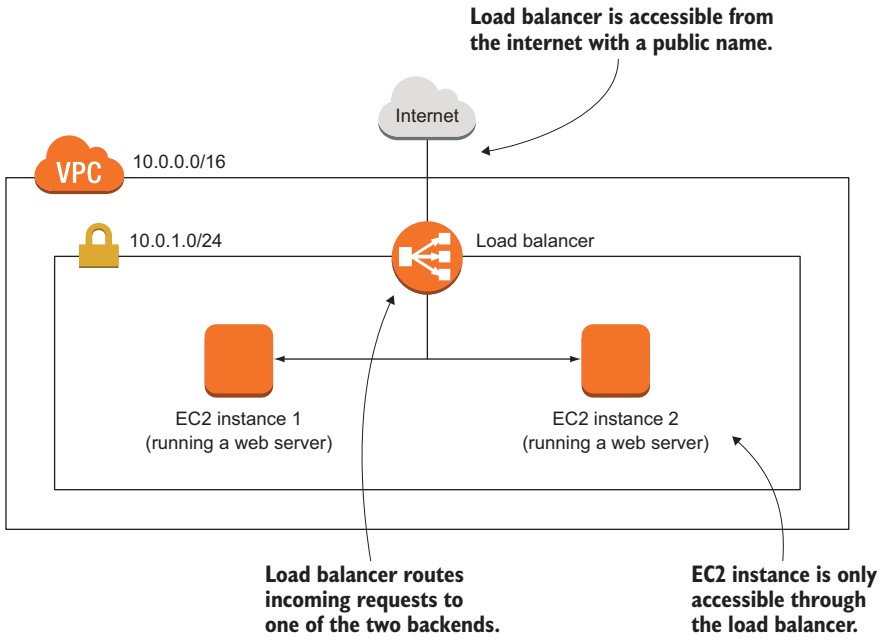


Figure 15.1 A load balancer synchronously decouples your EC2 instances.

The Classic Load Balancer is the oldest of the load balancers. If you start a new project, we recommend going with the ALB or NLB, because they are in most cases more cost efficient and more feature-rich.

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

Load balancers can be used with more than web servers—you can use load balancers in front of any systems that deal with request/response-style communication as long as the protocol is based on TCP.

15.1.1 Setting up a load balancer with virtual machines

AWS shines when it comes to integrating services. In chapter 14, you learned about auto-scaling groups. You'll now put an ALB in front of an auto-scaling group to decouple traffic to web servers, to remove a dependency between your users and the EC2 instance's public IP address. The auto-scaling group will make sure you always have two web servers running. As you learned in chapter 14, that's the way to protect against downtime caused by hardware failure. Servers that are started in the auto-scaling group will automatically register with the ALB. Figure 15.2 shows what the setup will look like. The interesting part is that the EC2 instances are no longer accessible directly from the public internet, so your users don't know about them. They don't know if there are 2 or 20 EC2 instances running behind the load balancer. Only the

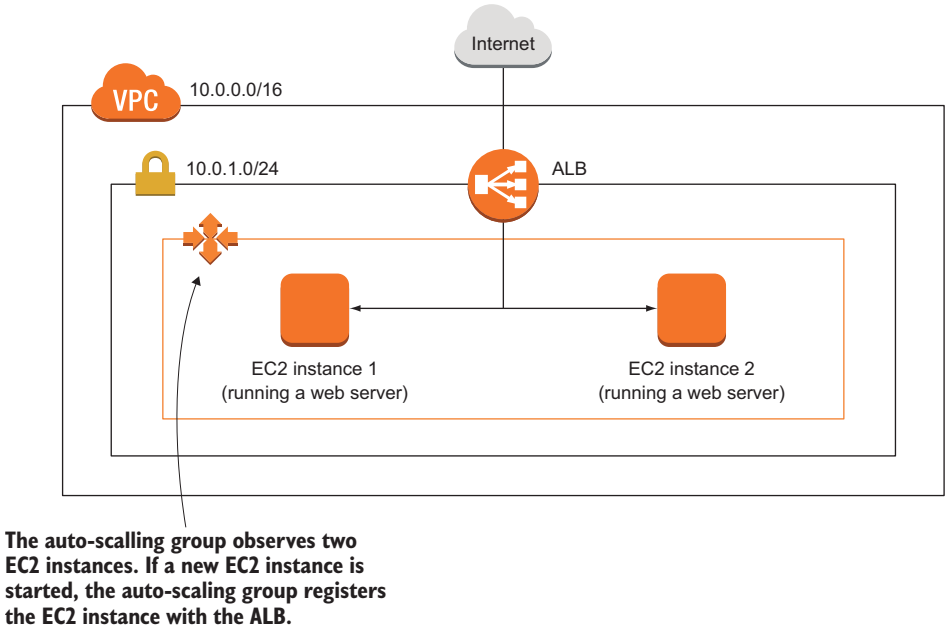


Figure 15.2 Auto-scaling groups work closely with ALB: they register a new web server with the load balancer.

load balancer is accessible and forwards requests to the backend servers behind it. The network traffic to load balancers and backend EC2 instances is controlled by security groups, which you learned about in chapter 6. If the auto-scaling group adds or removes EC2 instances, it will also register new EC2 instances with the load balancer and deregister EC2 instances that have been removed.

An ALB consists of three required parts and one optional part:

- *Load balancer*—Defines some core configurations, like the subnets the load balancer runs in, whether the load balancer gets public IP addresses, whether it uses IPv4 or both IPv4 and IPv6, and additional attributes.
- *Listener*—The listener defines the port and protocol that you can use to make requests to the load balancer. If you like, the listener can also terminate TLS for you. A listener links to a target group that is used as the default if no other listener rules match the request.
- *Target group*—A target group defines your group of backends. The target group is responsible for checking the backends by sending periodic health checks. Usually backends are EC2 instances, but could also be a Docker container running on EC2 Container Service or a machine in your data center paired with your VPC.

- *Listener rule*—Optional. You can define a listener rule. The rule can choose a different target group based on the HTTP path or host. Otherwise requests are forwarded to the default target group defined in the listener.

Figure 15.3 shows the ALB parts.

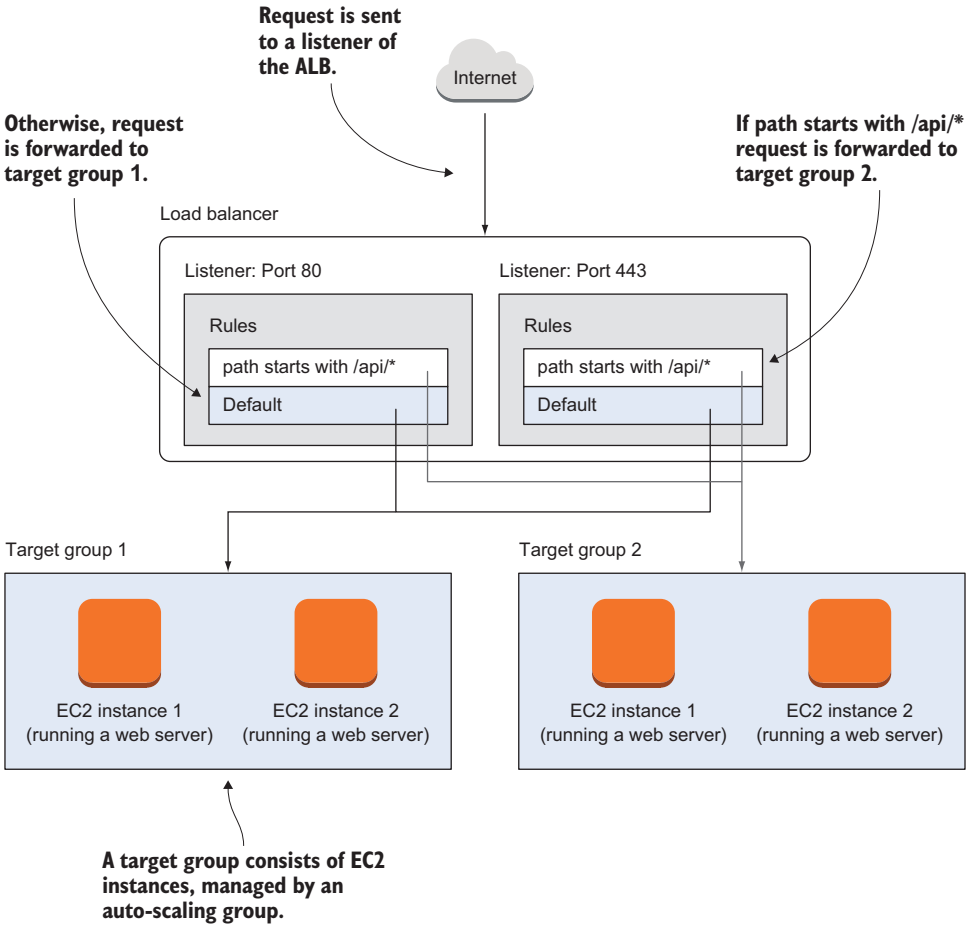
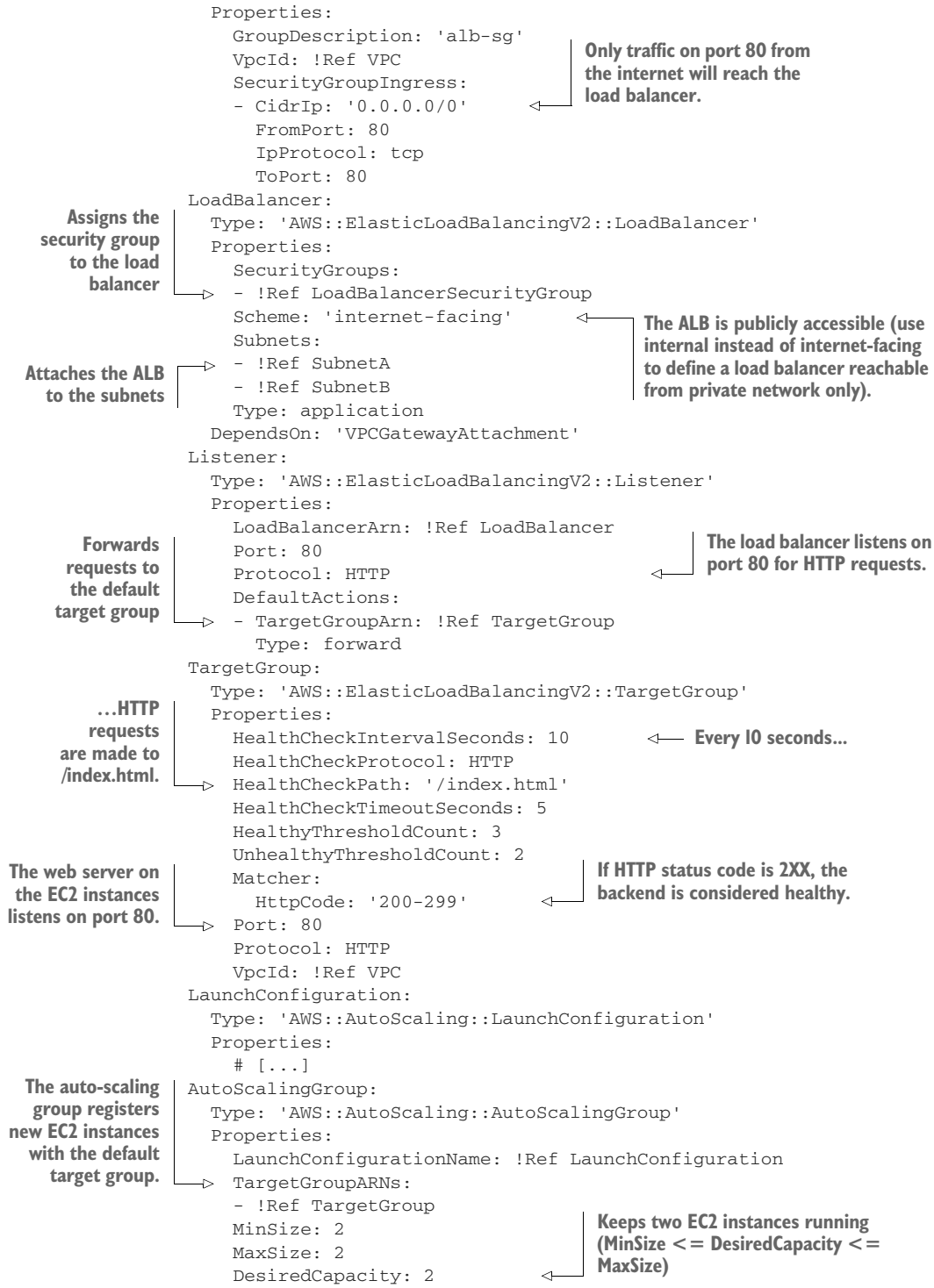


Figure 15.3 ALB consists of parts: load balancer, listener, target group, and optional listener rules.

The next listing shows a CloudFormation template snippet to create an ALB and connect it with an auto-scaling group. The listing implements the example shown in figure 15.2.

Listing 15.1 Creating a load balancer and connecting it to an auto-scaling group

```
# [...]
LoadBalancerSecurityGroup:
  Type: 'AWS::EC2::SecurityGroup'
```


Properties:
GroupDescription: 'alb-sg'
VpcId: !Ref VPC
SecurityGroupIngress:
- CidrIp: '0.0.0.0/0'
FromPort: 80
IpProtocol: tcp
ToPort: 80
Only traffic on port 80 from the internet will reach the load balancer.
LoadBalancer:
Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
Properties:
SecurityGroups:
- !Ref LoadBalancerSecurityGroup
Assigns the security group to the load balancer
Scheme: 'internet-facing'
The ALB is publicly accessible (use internal instead of internet-facing to define a load balancer reachable from private network only).
Subnets:
- !Ref SubnetA
- !Ref SubnetB
Attaches the ALB to the subnets
Type: application
DependsOn: 'VPCGatewayAttachment'
Listener:
Type: 'AWS::ElasticLoadBalancingV2::Listener'
Properties:
LoadBalancerArn: !Ref LoadBalancer
Port: 80
Protocol: HTTP
The load balancer listens on port 80 for HTTP requests.
DefaultActions:
- TargetGroupArn: !Ref TargetGroup
Type: forward
Forwards requests to the default target group
TargetGroup:
Type: 'AWS::ElasticLoadBalancingV2::TargetGroup'
Properties:
HealthCheckIntervalSeconds: 10
Every 10 seconds...
HealthCheckProtocol: HTTP
HealthCheckPath: '/index.html'
...HTTP requests are made to /index.html.
HealthCheckTimeoutSeconds: 5
HealthyThresholdCount: 3
UnhealthyThresholdCount: 2
Matcher:
HttpCode: '200-299'
If HTTP status code is 2XX, the backend is considered healthy.
Port: 80
The web server on the EC2 instances listens on port 80.
Protocol: HTTP
VpcId: !Ref VPC
LaunchConfiguration:
Type: 'AWS::AutoScaling::LaunchConfiguration'
Properties:
[...]
AutoScalingGroup:
Type: 'AWS::AutoScaling::AutoScalingGroup'
Properties:
LaunchConfigurationName: !Ref LaunchConfiguration
TargetGroupARNs:
- !Ref TargetGroup
The auto-scaling group registers new EC2 instances with the default target group.
MinSize: 2
MaxSize: 2
DesiredCapacity: 2
Keeps two EC2 instances running (MinSize <= DesiredCapacity <= MaxSize)

```
      VPCZoneIdentifier:
      - !Ref SubnetA
      - !Ref SubnetB
    DependsOn: 'VPCGatewayAttachment'
```

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

The full CloudFormation template is located at http://mng.bz/S6Sj. Create a stack based on that template by clicking on the Quick-Create link at http://mng.bz/lbG4, and then visit the output of your stack with your browser. Every time you reload the page, you should see one of the private IP addresses of a backend web server.

To get some detail about the load balancer in the graphical user interface, navigate to the EC2 Management Console. The sub-navigation menu on the left has a Load Balancing section where you can find a link to your load balancers. Select the one and only load balancer. You will see details at the bottom of the page. The details contain a Monitoring tab, where you can find charts about latency, number of requests, and much more. Keep in mind that those charts are one minute behind, so you may have to wait until you see the requests you made to the load balancer.

Cleaning up

Delete the CloudFormation stack you created.

15.2 Asynchronous decoupling with message queues

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

A message queue has a head and a tail. You can add new messages to the tail while reading messages from the head. This allows you to decouple the production and consumption of messages. Decoupling the producers/requesters from consumers/receivers delivers with the following benefits:

- *The queue acts as a buffer*—Producers and consumers don't have to run at the same speed. For example, you can add a batch of 1,000 messages in one minute while your consumers always process 10 messages per second. Sooner or later, the consumers will catch up and the queue will be empty again.
- *The queue hides your backend*—Similar to the load balancer, messages producers have no knowledge of the consumers. You can even stop all consumers and still produce messages. This is handy while doing maintenance on your consumers.

The producers and consumers don't know each other; they both only know about the message queue. Figure 15.4 illustrates this principle.

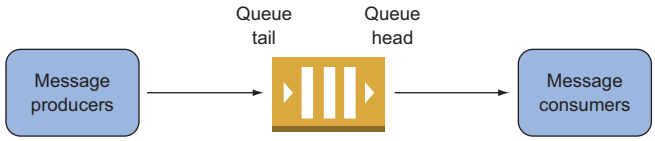


Figure 15.4 Producers send messages to a message queue while consumers read messages.

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

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

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

This limitation of SQS is also beneficial:

- You can put as many messages into SQS as you like.
- The message queue scales with the number of messages you produce and consume.
- SQS is highly available by default.
- You pay per message.

The pricing model is also simple: you pay $0.00000040 USD per request to SQS or $0.4 USD per million requests. Producing a message is one request, and consuming is another request (if your payload is larger than 64 KB, every 64 KB chunk counts as one request).

15.2.1 Turning a synchronous process into an asynchronous one

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

1. The user submits a URL.
2. The web server downloads the content at the URL, takes a screenshot, and renders it as a PNG image.
3. The web server returns the PNG to the user.

With one small trick, this process can be made asynchronous, and benefit from the elasticity of a message queue, for example during peak traffic:

1. The user submits a URL.
2. The web server puts a message into a queue that contains a random ID and the URL.
3. The web server returns a link to the user where the PNG image will be found in the future. The link contains the random ID (such as http://$Bucket.s3-website-us-east-1.amazonaws.com/$RandomId.png).
4. In the background, a worker consumes the message from the queue, downloads the content, converts the content into a PNG, and uploads the image to S3.
5. At some point, the user tries to download the PNG at the known location. If the file is not found, the user should reload the page in a few seconds.

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

15.2.2 Architecture of the URL2PNG application

You'll now create a simple but decoupled piece of software named URL2PNG that renders a PNG from a given web URL. You'll use Node.js to do the programming part, and you'll use SQS as the message queue implementation. Figure 15.5 shows how the URL2PNG application works.

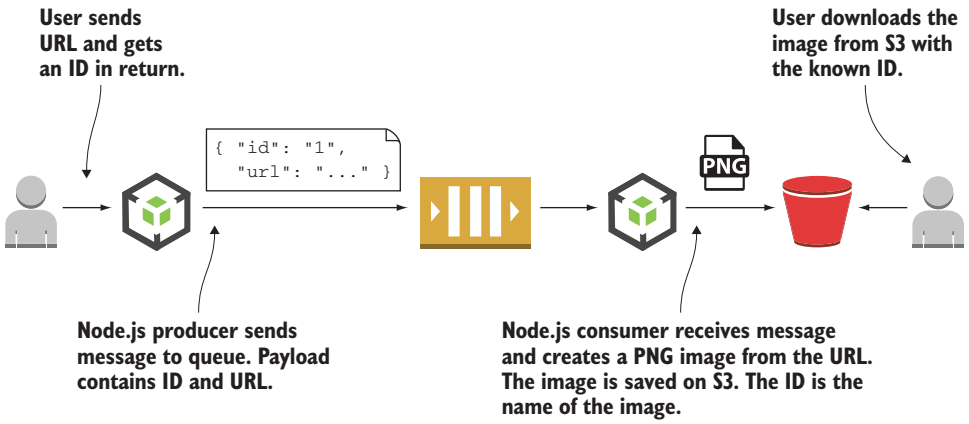


Figure 15.5 Node.js producer sends a message to the queue. The payload contains an ID and URL.

On the message producer side, a small Node.js scripts generates a unique ID, sends a message to the queue with the URL and ID as the payload, and returns the ID to the user. The user now starts checking if a file is available on the S3 bucket using the returned ID as the filename.

Simultaneously, on the message consumer side, a small Node.js script reads a message from the queue, generates the screenshot of the URL from the payload, and uploads the resulting image to an S3 bucket using the unique ID from the payload as the filename.

To complete the example, you need to create an S3 bucket with web hosting enabled. Execute the following commands, replacing *$yourname* with your name or nickname to prevent name clashes with other readers (remember that S3 bucket names have to be globally unique across all AWS accounts):

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

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

15.2.3 Setting up a message queue

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

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

The returned `QueueUrl` is needed later in the example, so take a note.

15.2.4 Producing messages programmatically

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

> **Installing and getting started with Node.js**
>
> Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.
>
> After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v8.*`. Now you're ready to run JavaScript examples like URL2PNG.
>
> Do you want to get started with Node.js? We recommend *Node.js in Action (2nd edition)* from Alex Young, et al., (Manning, 2017) or the video course *Node.js in Motion* from P.J. Evans, (Manning, 2018).

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

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

As usual, you'll find the code in the book's code repository on GitHub https://github.com/AWSinAction/code2. The URL2PNG example is located at /chapter15/url2png/. Here's the implementation of index.js.

Listing 15.2 index.js: sending a message to the queue

```
const AWS = require('aws-sdk');
const uuid = require('uuid/v4');
const sqs = new AWS.SQS({                          <-- Creates an SQS client
  region: 'us-east-1'
});

if (process.argv.length !== 3) {                   <-- Checks whether a URL was provided
  console.log('URL missing');
  process.exit(1);
}

const id = uuid();                                 <-- Creates a random ID
const body = {                                     <-- The payload contains the random ID and the URL.
  id: id,
  url: process.argv[2]
};

sqs.sendMessage({                                  <-- Converts the payload into a JSON string
  MessageBody: JSON.stringify(body),
  QueueUrl: '$QueueUrl'                            <-- Queue to which the message is sent (was returned when creating the queue).
}, (err) => {                                      <-- Invokes the sendMessage operation on SQS
  if (err) {
    console.log('error', err);
  } else {
    console.log('PNG will be soon available at ...');
  }
});
```

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

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

```
$ aws sqs get-queue-attributes \
➥ --queue-url "$QueueUrl" \
➥ --attribute-names ApproximateNumberOfMessages
```

```
{
  "Attributes": {
    "ApproximateNumberOfMessages": "1"
  }
}
```

SQS only returns an approximation of the number of messages. This is due to the distributed nature of SQS. If you don't see your message in the approximation, run the command again and eventually you will see your message.

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

15.2.5 Consuming messages programmatically

Processing a message with SQS takes three steps:

1. Receive a message.
2. Process the message.
3. Acknowledge that the message was successfully processed.

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

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

- `QueueUrl`—The unique queue identifier.
- `MaxNumberOfMessages`—The maximum number of messages you want to receive (from 1 to 10). To get higher throughput, you can get messages in a batch. We usually set this to 10 for best performance and lowest overhead.
- `VisibilityTimeout`—The number of seconds you want to remove this message from the queue to process it. Within that time, you must delete the message, or it will be delivered back to the queue. We usually set this to the average processing time multiplied by four.
- `WaitTimeSeconds`—The maximum number of seconds you want to wait to receive messages if they're not immediately available. Receiving messages from SQS is done by polling the queue. But AWS allows long polling, for a maximum of 10 seconds. When using long polling, you will not get an immediate response from the AWS API if no messages are available. If a new message arrives within 10 seconds, the HTTP response will be sent to you. After 10 seconds, you also get an empty response.

Ths listing shows how this is done with the SDK.

Listing 15.3 worker.js: receiving a message from the queue

```
const fs = require('fs');
const AWS = require('aws-sdk');
const webshot = require('webshot');
const sqs = new AWS.SQS({
  region: 'us-east-1'
```

```
});
const s3 = new AWS.S3({
  region: 'us-east-1'
});

const receive = (cb) => {
  const params = {
    QueueUrl: '$QueueUrl',
    MaxNumberOfMessages: 1,
    VisibilityTimeout: 120,
    WaitTimeSeconds: 10
  };
  sqs.receiveMessage(params, (err, data) => {
    if (err) {
      cb(err);
    } else {
      if (data.Messages === undefined) {
        cb(null, null);
      } else {
        cb(null, data.Messages[0]);
      }
    }
  });
};
```

Consumes no more than one message at once

Takes the message from the queue for 120 seconds

Long poll for 10 seconds to wait for new messages

Invokes the receiveMessage operation on SQS

Checks whether a message is available

Gets the one and only message

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

Listing 15.4 worker.js: processing a message (take screenshot and upload to S3)

```
const process = (message, cb) => {
  const body = JSON.parse(message.Body);
  const file = body.id + '.png';
  webshot(body.url, file, (err) => {
    if (err) {
      cb(err);
    } else {
      fs.readFile(file, (err, buf) => {
        if (err) {
          cb(err);
        } else {
          const params = {
            Bucket: 'url2png-$yourname',
            Key: file,
            ACL: 'public-read',
            ContentType: 'image/png',
            Body: buf
          };
          s3.putObject(params, (err) => {
            if (err) {
              cb(err);
            } else {
              fs.unlink(file, cb);
            }
```

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

Creates the screenshot with the webshot module

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

Allows everyone to read the screenshot on S3

Uploads the screenshot to S3

Removes the screenshot from local disk

```
          });
        }
      });
    }
  });
};
```

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

Listing 15.5 worker.js: acknowledging a message (deletes the message from the queue)

```
const acknowledge = (message, cb) => {
  const params = {
    QueueUrl: '$QueueUrl',
    ReceiptHandle: message.ReceiptHandle
  };
  sqs.deleteMessage(params, cb);
};
```

ReceiptHandle is unique for each receipt of a message.

Invokes the deleteMessage operation on SQS

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

Listing 15.6 worker.js: connecting the parts

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

Receives a message

Checks whether a message is available

Calls the run method again in one second

Processes the message

Acknowledges the message

Calls the run method again in one second to poll for further messages (kind of a recursive loop, but with a timer in between. When the timer starts, a new call stack is allocated; this will not lead to a stack overflow!).

```
  });
};

run();                    <-- Calls the run method to start
```

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

Remember the output you got when you invoked `node index.js "http://aws.amazon.com"` to send a message to the queue? It looked similar to this: http://url2png-$yourname.s3-website-us-east-1.amazonaws.com/XYZ.png. Now put that URL in your web browser and you will find a screenshot of the AWS website (or whatever you used as an example).

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

Cleaning up

Delete the message queue as follows:

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

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

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

15.2.6 Limitations of messaging with SQS

Earlier in the chapter, we mentioned a few limitations of SQS. This section covers them in more detail. But before we start with the limitations, here are the benefits:

- You can put as many messages into SQS as you like. SQS scales the underlying infrastructure for you.
- SQS is highly available by default.
- You pay per message.

Those benefits come with some trade-offs. Let's have a look of the limitations in more detail now.

SQS DOESN'T GUARANTEE THAT A MESSAGE IS DELIVERED ONLY ONCE

There are two reasons why a message might be delivered more than once:

1. Common reason: If a received message isn't deleted within `VisibilityTimeout`, the message will be received again.
2. Rare reason: If a `DeleteMessage` operation doesn't delete all copies of a message because one of the servers in the SQS system isn't available at the time of deletion.

The problem of repeated delivery of a message can be solved by making the message processing idempotent. *Idempotent* means that no matter how often the message is processed, the result stays the same. In the URL2PNG example, this is true by design: If you process the message multiple times, the same image will be uploaded to S3 multiple times. If the image is already available on S3, it's replaced. Idempotence solves many problems in distributed systems that guarantee messages will be delivered at least once.

Not everything can be made idempotent. Sending an email is a good example: if you process a message multiple times and it sends an email each time, you'll annoy the addressee.

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

SQS DOESN'T GUARANTEE THE MESSAGE ORDER

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

> **SQS FIFO (first-in-first-out) queues**
>
> FIFO queues guarantee order of messages and have a mechanism to detect duplicate messages. If you need a strict message order, they are worth a look. The disadvantages are higher pricing and a limitation on 300 operations per second. Check out the documentation at http://mng.bz/Y5KN for more information.

SQS DOESN'T REPLACE A MESSAGE BROKER

SQS isn't a message broker like ActiveMQ—SQS is only a message queue. Don't expect features like message routing or message priorities. Comparing SQS to ActiveMQ is like comparing DynamoDB to MySQL.

Amazon MQ

AWS announced an alternative to Amazon SQS in November 2017: Amazon MQ provides Apache ActiveMQ as a service. Therefore, you can use Amazon MQ as a message broker that speaks the JMS, NMS, AMQP, STOMP, MQTT, and WebSocket protocols.

Go to the Amazon MQ Developer Guide at https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/ to learn more.

Summary

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

16 Designing for fault tolerance

This chapter covers

- What is fault-tolerance and why do you need it?
- Using redundancy to remove single points of failure
- Retrying on failure
- Using idempotent operations to achieve retry on failure
- AWS service guarantees

Failure is inevitable: hard disks, networks, power, and so on all fail from time to time. Fault tolerance deals with that problem. A fault-tolerant architecture is built for failure. If a failure occurs, the system isn't interrupted, and it continues to handle requests. If there is single point of failure within your architecture, it is not fault-tolerant. You can achieve fault-tolerance by introducing redundancy into your system and by decoupling the parts of your architecture such that one side does not rely on the uptime of the other.

The services provided by AWS offer different types of failure resilience:

- *No guarantees (single point of failure)*—No requests are served in case of failure.
- *High availability*—In case of failure, it takes some time until requests are served as before.
- *Fault-tolerance*—In case of failure, requests are served as before without any availability issues.

The most convenient way to make your system fault-tolerant is to build the architecture using fault-tolerant blocks. If all blocks are fault-tolerant, the whole system will be fault-tolerant as well. Luckily, many AWS services are fault-tolerant by default. If possible, use them. Otherwise you'll need to deal with the consequences and handle failures yourself.

Unfortunately, one important service isn't fault-tolerant by default: EC2 instances. Virtual machines aren't fault-tolerant. This means an architecture that uses EC2 isn't fault-tolerant by default. But AWS provides the building blocks to deal with that issue. The solution consists of auto-scaling groups, Elastic Load Balancing (ELB), and Simple Queue Service (SQS).

The following services provided by AWS are neither highly available nor fault-tolerant. When using one of these services in your architecture, you are adding a *single point of failure* (SPOF) to your infrastructure. In this case, to achieve fault-tolerance, you need to plan and build for failure as discussed during the rest of the chapter.

- *Amazon Elastic Compute Cloud (EC2) instance*—A single EC2 instance can fail for many reasons: hardware failure, network problems, availability-zone outage, and so on. To achieve high availability or fault-tolerance, use auto-scaling groups to set up a fleet of EC2 instances that serve requests in a redundant way.
- *Amazon Relational Database Service (RDS) single instance*—A single RDS instance could fail for the same reasons than an EC2 instance might fail. Use Multi-AZ mode to achieve high availability.

All the following services are *highly available* (HA) by default. When a failure occurs, the services will suffer from a short downtime but will recover automatically:

- *Elastic Network Interface (ENI)*—A network interface is bound to an AZ, so if this AZ goes down, your network interface will be unavailable as well.
- *Amazon Virtual Private Cloud (VPC) subnet*—A VPC subnet is bound to an AZ, so if this AZ suffers from an outage, your subnet will not be reachable as well. Use multiple subnets in different AZs to remove the dependency on a single AZ.
- *Amazon Elastic Block Store (EBS) volume*—An EBS volume distributes data among multiple machines within an AZ. But if the whole AZ fails, your volume will be unavailable (you won't lose your data though). You can create EBS snapshots from time to time so you can re-create an EBS volume in another AZ.
- *Amazon Relational Database Service (RDS) Multi-AZ instance*—When running in Multi-AZ mode, a short downtime (1 minute) is expected if an issue occurs with the master instance while changing DNS records to switch to the standby instance.

The following services are *fault-tolerant* by default. As a consumer of the service, you won't notice any failures.

- Elastic Load Balancing (ELB), deployed to at least two AZs
- Amazon EC2 security groups
- Amazon Virtual Private Cloud (VPC) with an ACL and a route table
- Elastic IP addresses (EIP)
- Amazon Simple Storage Service (S3)
- Amazon Elastic Block Store (EBS) snapshots
- Amazon DynamoDB
- Amazon CloudWatch
- Auto-scaling groups
- Amazon Simple Queue Service (SQS)
- AWS Elastic Beanstalk—The management service itself, not necessarily your application running within the environment
- AWS OpsWorks—The management service itself, not necessarily your application running within the environment
- AWS CloudFormation
- AWS Identity and Access Management (IAM, not bound to a single region; if you create an IAM user, that user is available in all regions)

Why should you care about fault tolerance? Because in the end, a fault-tolerant system provides the highest quality to your end users. No matter what happens in your system, the user is never affected and can continue to consume entertaining content, buy goods and services, or have conversations with friends. A few years ago, achieving fault tolerance was expensive and complicated, but in AWS, providing fault-tolerant systems is becoming an affordable standard. Nevertheless, building fault-tolerant systems is the supreme discipline of cloud computing, and might be challenging at the beginning.

Chapter requirements

To fully understand this chapter, you need to have read and understood the following concepts:

- EC2 (chapter 3)
- Auto-scaling (chapter 14)
- Elastic Load Balancing (chapter 15)
- Simple Queue Service (chapter 15)

On top of that, the example included in this chapter makes intensive use of the following:

- Elastic Beanstalk (chapter 5)
- DynamoDB (chapter 13)
- Express, a Node.js web application framework

In this chapter, you'll learn everything you need to design a fault-tolerant web application based on EC2 instances (which aren't fault-tolerant by default).

16.1 Using redundant EC2 instances to increase availability

Here are just a few reasons why your virtual machine might fail:

- If the host hardware fails, it can no longer host the virtual machine on top of it.
- If the network connection to/from the host is interrupted, the virtual machine will lose the ability to communicate over network.
- If the host system is disconnected from the power supply, the virtual machine will fail as well.

Additionally the software running inside your virtual machine may also cause a crash:

- If your application contains a memory leak, you'll run out of memory and fail. It may take a day, a month, a year, or more, but eventually it will happen.
- If your application writes to disk and never deletes its data, you'll run out of disk space sooner or later, causing your application to fail.
- Your application may not handle edge cases properly and may instead crash unexpectedly.

Regardless of whether the host system or your application is the cause of a failure, a single EC2 instance is a single point of failure. If you rely on a single EC2 instance, your system will blow up eventually. It's merely a matter of time.

16.1.1 Redundancy can remove a single point of failure

Imagine a production line that makes fluffy cloud pies. Producing a fluffy cloud pie requires several production steps (simplified!):

1. Produce a pie crust.
2. Cool the pie crust.
3. Put the fluffy cloud mass on top of the pie crust.
4. Cool the fluffy cloud pie.
5. Package the fluffy cloud pie.

The current setup is a single production line. The big problem with this setup is that whenever one of the steps crashes, the entire production line must be stopped. Figure 16.1 illustrates the problem when the second step (cooling the pie crust) crashes. The steps that follow no longer work either, because they no longer receive cool pie crusts.

Why not have multiple production lines? Instead of one line, suppose we have three. If one of the lines fails, the other two can still produce fluffy cloud pies for all the hungry customers in the world. Figure 16.2 shows the improvements; the only downside is that we need three times as many machines.

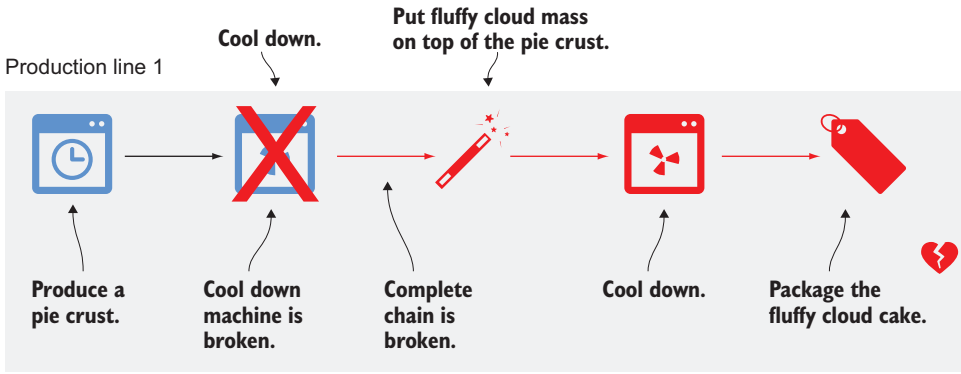
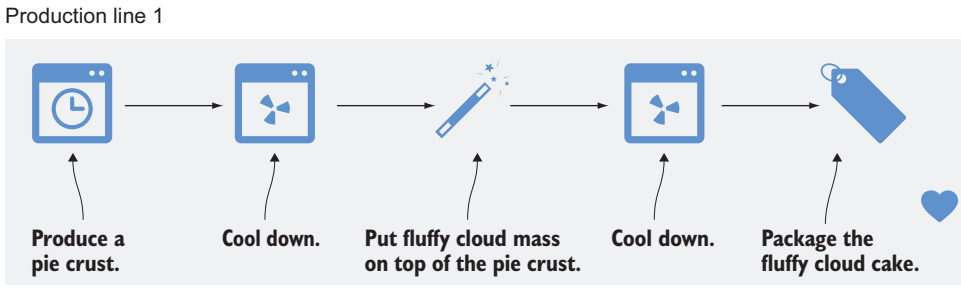
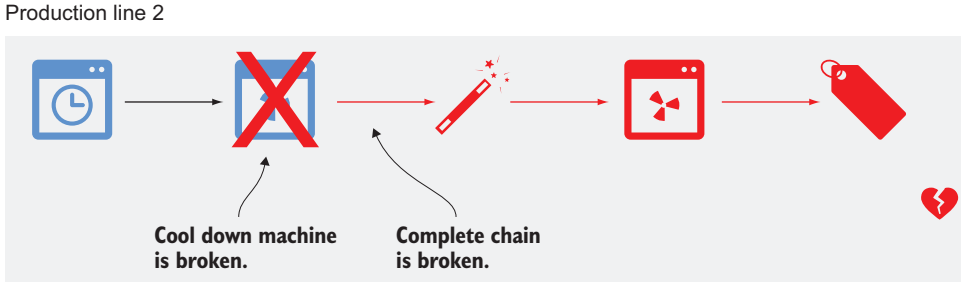


Figure 16.1 A single point of failure affects not only itself, but the entire system.





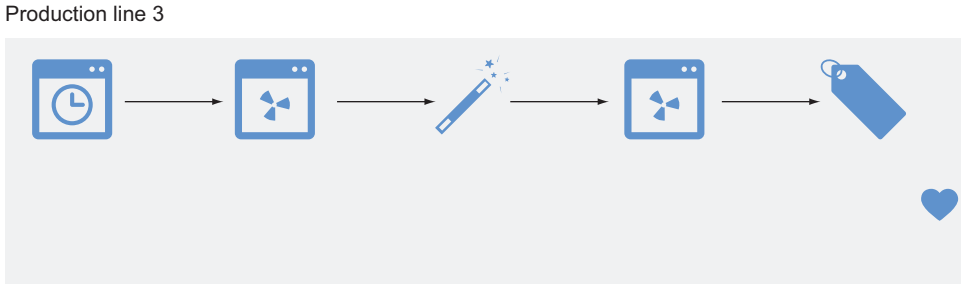


Figure 16.2 Redundancy eliminates single points of failure and makes the system more stable.

The example can be transferred to EC2 instances as well. Instead of having only one EC2 instance running your application, you can have three. If one of those instances fails, the other two will still be able to serve incoming requests. You can also minimize the cost impact of one versus three instances: instead of one large EC2 instance, you can choose three small ones. The problem that arises with a dynamic EC2 instance pool is: how can you communicate with the instances? The answer is *decoupling*: put a load balancer or message queue between your EC2 instances and the requester. Read on to learn how this works.

16.1.2 Redundancy requires decoupling

Figure 16.3 shows how EC2 instances can be made fault-tolerant by using redundancy and synchronous decoupling. If one of the EC2 instances crashes, the Elastic Load Balancer (ELB) stops routing requests to the crashed instances. The auto-scaling group replaces the crashed EC2 instance within minutes, and the ELB begins to route requests to the new instance.

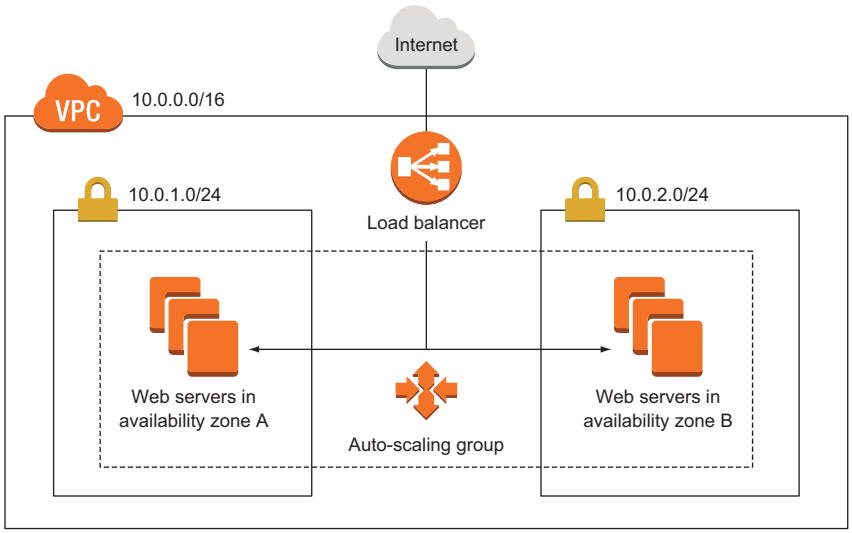


Figure 16.3 Fault-tolerant EC2 instances with an auto-scaling group and an Elastic Load Balancer

Take a second look at figure 16.3 and see what parts are redundant:

- *Availability zones (AZs)*—Two are used. If one AZ suffers from an outage, we still have instances running in the other AZ.
- *Subnets*—A subnet is tightly coupled to an AZ. Therefore we need one subnet in each AZ.
- *EC2 instances*—We have multi-redundancy for EC2 instances. We have multiple instances in one subnet (AZ), and we have instances in two subnets (AZs).

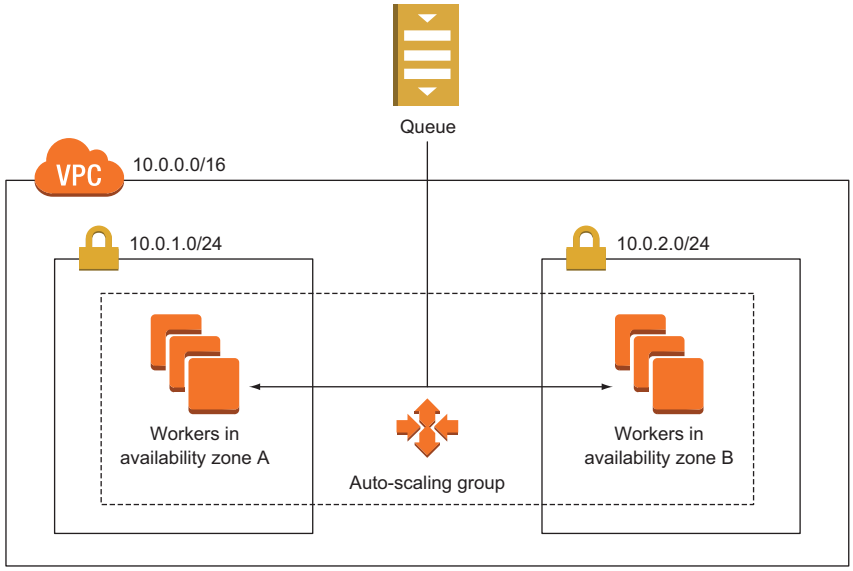


Figure 16.4 Fault-tolerant EC2 instances with an auto-scaling group and SQS

Figure 16.4 shows a fault-tolerant system built with EC2 that uses the power of redundancy and asynchronous decoupling to process messages from an SQS queue.

In both figures, the load balancer and the SQS queue appear only once. This doesn't mean that ELB or SQS are single points of failure; on the contrary, ELB and SQS are both fault-tolerant by default.

16.2 Considerations for making your code fault-tolerant

If you want to achieve fault tolerance, you have to build your application accordingly. You can design fault tolerance into your application by following two suggestions presented in this section.

16.2.1 Let it crash, but also retry

The Erlang programming language is famous for the concept of "let it crash." That means whenever the program doesn't know what to do, it crashes, and someone needs to deal with the crash. Most often people overlook the fact that Erlang is also famous for retrying. Letting it crash without retrying isn't useful—if you can't recover from a crash, your system will be down, which is the opposite of what you want.

You can apply the "let it crash" concept (some people call it "fail fast") to synchronous and asynchronous decoupled scenarios. In a synchronous decoupled scenario, the sender of a request must implement the retry logic. If no response is returned within a certain amount of time, or an error is returned, the sender retries by sending the same request again. In an asynchronous decoupled scenario, things are easier. If a message is consumed but not acknowledged within a certain amount of time, it goes

back to the queue. The next consumer then grabs the message and processes it again. Retrying is built into asynchronous systems by default.

"Let it crash" isn't useful in all situations. If the program wants to respond to the sender that the request contained invalid content, this isn't a reason for letting the server crash: the result will stay the same no matter how often you retry. But if the server can't reach the database, it makes a lot of sense to retry. Within a few seconds, the database may be available again and able to successfully process the retried request.

Retrying isn't that easy. Imagine that you want to retry the creation of a blog post. With every retry, a new entry in the database is created, containing the same data as before. You end up with many duplicates in the database. Preventing this involves a powerful concept that's introduced next: idempotent retry.

16.2.2 Idempotent retry makes fault tolerance possible

How can you prevent a blog post from being added to the database multiple times because of a retry? A naïve approach would be to use the title as primary key. If the primary key is already used, you can assume that the post is already in the database and skip the step of inserting it into the database. Now the insertion of blog posts is *idempotent*, which means no matter how often a certain action is applied, the outcome must be the same. In the current example, the outcome is a database entry.

It continues with a more complicated example. Inserting a blog post is more complicated in reality, as the process might look something like this:

1. Create a blog post entry in the database.
2. Invalidate the cache because data has changed.
3. Post the link to the blog's Twitter feed.

Let's take a close look at each step.

1. CREATING A BLOG POST ENTRY IN THE DATABASE

We covered this step earlier by using the title as a primary key. But this time, we use a universally unique identifier (UUID) instead of the title as the primary key. A UUID like `550e8400-e29b-11d4-a716-446655440000` is a random ID that's generated by the client. Because of the nature of a UUID, it's unlikely that two identical UUIDs will be generated. If the client wants to create a blog post, it must send a request to the ELB containing the UUID, title, and text. The ELB routes the request to one of the backend servers. The backend server checks whether the primary key already exists. If not, a new record is added to the database. If it exists, the insertion continues. Figure 16.5 shows the flow.

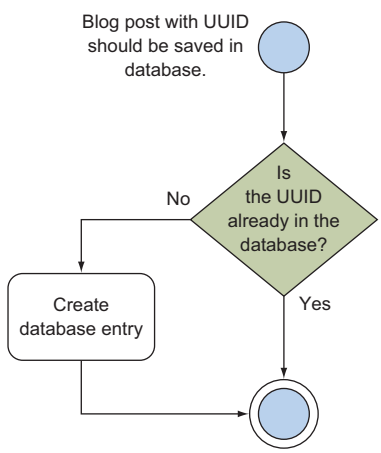


Figure 16.5 Idempotent database insert: creating a blog post entry in the database only if it doesn't already exist

Creating a blog post is a good example of an idempotent operation that is guaranteed by code. You can also use your database to handle this problem. Just send an insert to your database. Three things could happen:

1. *Your database inserts the data.* The operation is successfully completed.
2. *Your database responds with an error because the primary key is already in use.* The operation is successfully completed.
3. *Your database responds with a different error.* The operation crashes.

Think twice about the best way to implement idempotence!

2. Invalidating the cache

This step sends an invalidation message to a caching layer. You don't need to worry about idempotence too much here: it doesn't hurt if the cache is invalidated more often than needed. If the cache is invalidated, then the next time a request hits the cache, the cache won't contain data, and the original source (in this case, the database) will be queried for the result. The result is then put in the cache for subsequent requests. If you invalidate the cache multiple times because of a retry, the worst thing that can happen is that you may need to make a few more calls to your database. That's easy.

3. Posting to the blog's Twitter feed

To make this step idempotent, you need to use some tricks, because you interact with a third party that doesn't support idempotent operations. Unfortunately, no solution will guarantee that you post exactly one status update to Twitter. You can guarantee the creation of at least one (one or more than one) status update, or at most one (one or none) status update. An easy approach could be to ask the Twitter API for the latest status updates; if one of them matches the status update that you want to post, you skip the step because it's already done.

But Twitter is an eventually consistent system: there is no guarantee that you'll see a status update immediately after you post it. Therefore, you can end up having your status update posted multiple times. Another approach would be to save in a database whether you already posted the status update. But imagine saving to the database that you posted to Twitter and then making the request to the Twitter API—but at that moment, the system crashes. Your database will state that the Twitter status update was posted, but in reality it wasn't. You need to make a choice: tolerate a missing status update, or tolerate multiple status updates. Hint: it's a business decision. Figure 16.6 shows the flow of both solutions.

Now it's time for a practical example! You'll design, implement, and deploy a distributed, fault-tolerant web application on AWS. This example will demonstrate how distributed systems work and will combine most of the knowledge in this book.

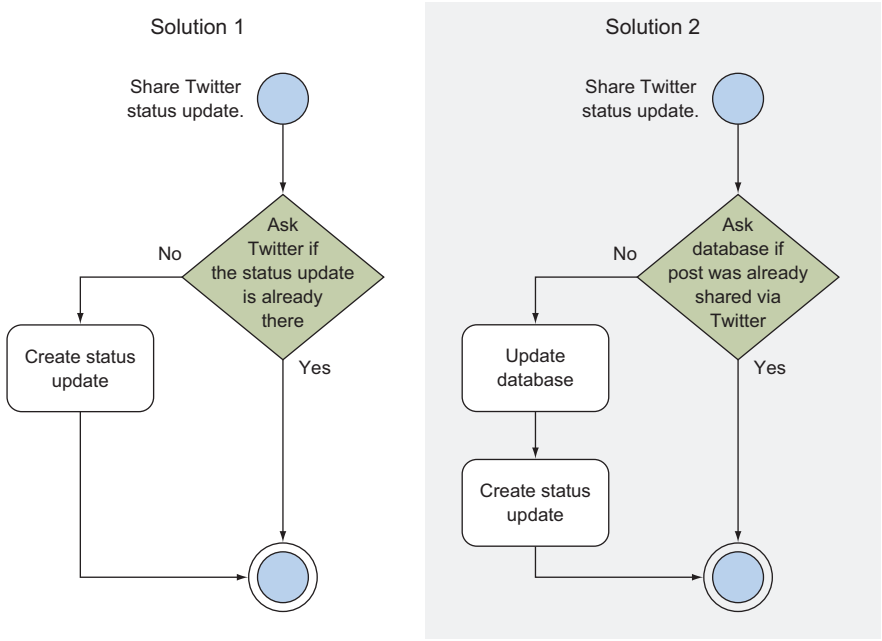


Figure 16.6 Idempotent Twitter status update: only share a status update if it hasn't already been done.

16.3 Building a fault-tolerant web application: Imagery

Before you begin the architecture and design of the fault-tolerant Imagery application, we'll talk briefly about what the application should do. A user should be able to upload an image. This image is then transformed with a sepia filter so that it looks old. The user can then view the sepia image. Figure 16.7 shows the process.

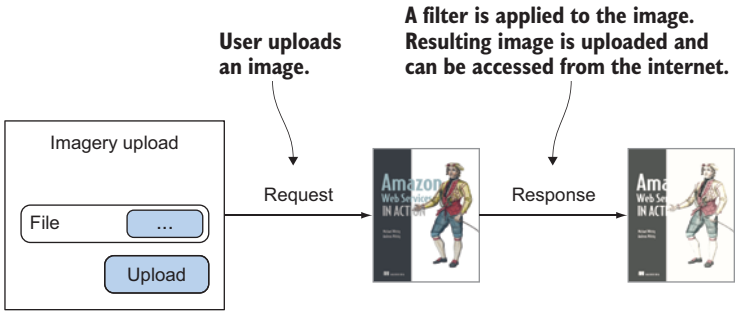


Figure 16.7 The user uploads an image to Imagery, where a filter is applied.

The problem with the process shown in figure 16.7 is that it's synchronous. If the web server dies during request and response, the user's image won't be processed. Another problem arises when many users want to use the Imagery app: the system

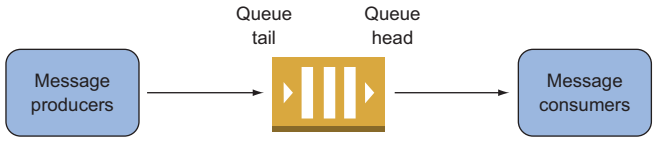


Figure 16.8 Producers send messages to a message queue while consumers read messages.

becomes busy and may slow down or stop working. Therefore the process should be turned into an asynchronous one. Chapter 15 introduced the idea of asynchronous decoupling by using an SQS message queue, as shown in figure 16.8.

When designing an asynchronous process, it's important to keep track of the process. You need some kind of identifier for it. When a user wants to upload an image, the user creates a process first. This returns a unique ID. With that ID, the user can upload an image. If the image upload is finished, the worker begins to process the image in the background. The user can look up the process at any time with the process ID. While the image is being processed, the user can't see the sepia image. But as soon as the image is processed, the lookup process returns the sepia image. Figure 16.9 shows the asynchronous process.

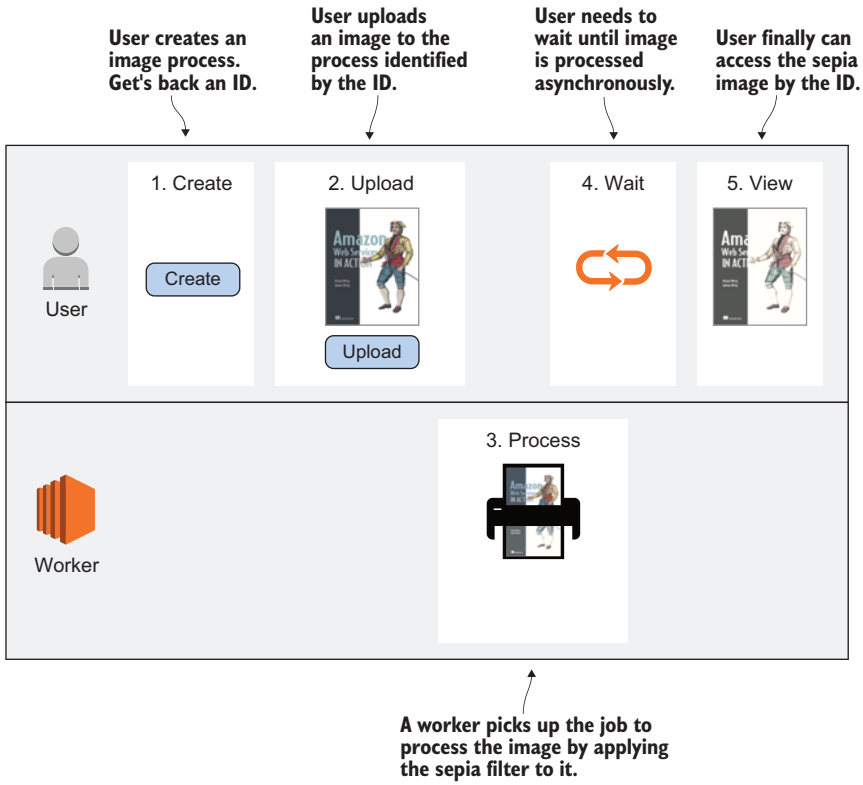


Figure 16.9 The user asynchronously uploads an image to Imagery, where a filter is applied.

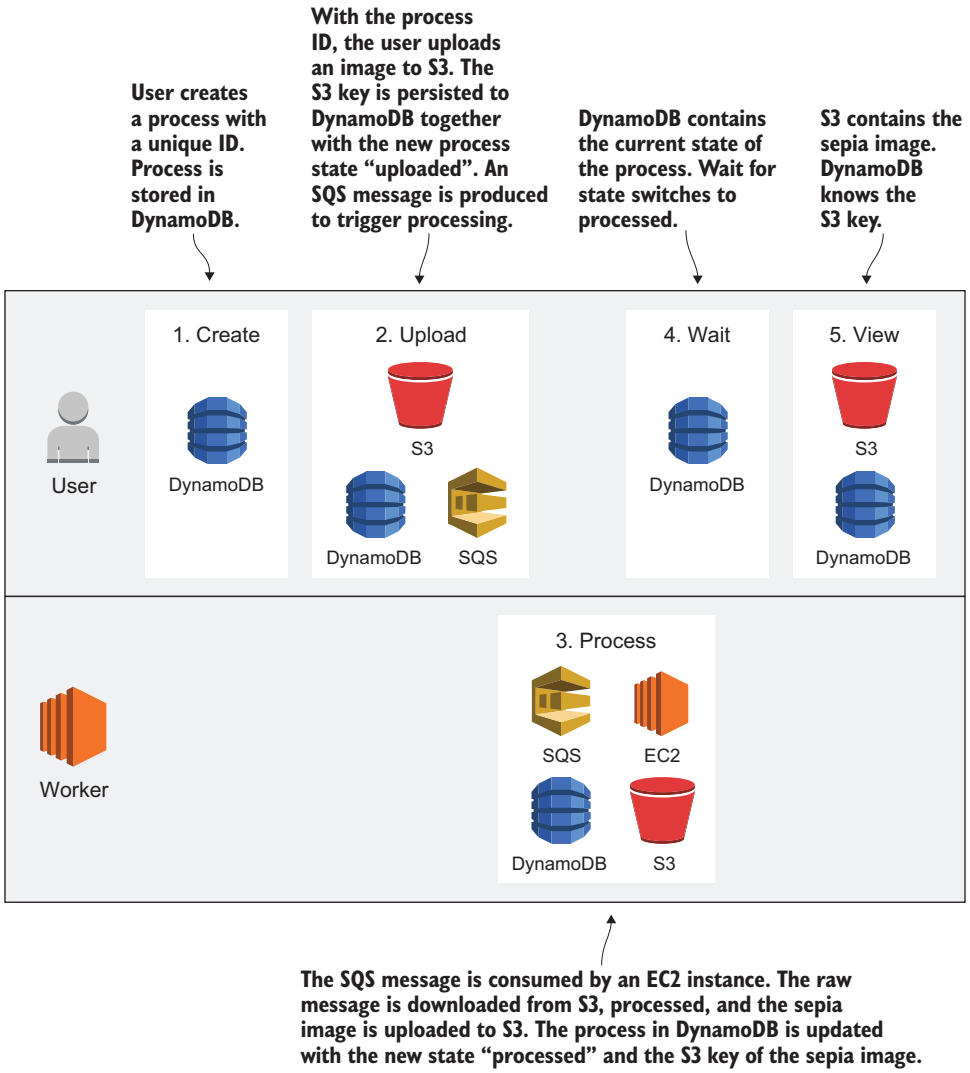


Figure 16.10 Combining AWS services to implement the asynchronous Imagery process

Now that you have an asynchronous process, it's time to map that process to AWS services. Keep in mind that most services on AWS are fault-tolerant by default, so it makes sense to pick them whenever possible. Figure 16.10 shows one way of doing it.

To make things as easy as possible, all the actions will be accessible via a REST API, which will be provided by EC2 instances. In the end, EC2 instances will provide the process and make calls to all the AWS services shown in figure 16.10.

You'll use many AWS services to implement the Imagery application. Most of them are fault-tolerant by default, but EC2 isn't. You'll deal with that problem using an idempotent state machine, as introduced in the next section.

Example is 100% covered by the Free Tier

The example in this chapter is totally covered by the Free Tier. As long as you don't run the example longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the example within a few days, because you'll clean up your account at the end of the section.

16.3.1 The idempotent state machine

An idempotent state machine sounds complicated. We'll take some time to explain it because it's the heart of the Imagery application. Let's look at what a *state machine* is and what *idempotent* means in this context.

THE FINITE STATE MACHINE

A finite state machine has at least one start state and one end state. Between the start and the end state, the state machine can have many other states. The machine also defines transitions between states. For example, a state machine with three states could look like this:

```
(A) -> (B) -> (C).
```

This means:

- State A is the start state.
- There is a transition possible from state A to B.
- There is a transition possible from state B to C.
- State C is the end state.

But there is no transition possible between (A) -> (C) or (B) -> (A). With this in mind, we apply the theory to our Imagery example. The Imagery state machine could look like this:

```
(Created) -> (Uploaded) -> (Processed)
```

Once a new process (state machine) is created, the only transition possible is to `Uploaded`. To make this transition happen, you need the S3 key of the uploaded raw image. So the transition between `Created` -> `Uploaded` can be defined by the function `uploaded(s3Key)`. Basically, the same is true for the transition `Uploaded` -> `Processed`. This transition can be done with the S3 key of the sepia image: `processed(s3Key)`.

Don't be confused by the fact that the upload and the image filter processing don't appear in the state machine. These are the basic actions that happen, but we're only interested in the results; we don't track the progress of the actions. The process isn't aware that 10% of the data has been uploaded or 30% of the image processing is done. It only cares whether the actions are 100% done. You can probably imagine a

bunch of other states that could be implemented, but we're skipping that for the purpose of simplicity in this example: resized and shared are just two examples.

IDEMPOTENT STATE TRANSITIONS

An idempotent state transition must have the same result no matter how often the transition takes place. If you can make sure that your state transitions are idempotent, you can do a simple trick: in case of a failure during transitioning, you retry the entire state transition.

Let's look at the two state transitions you need to implement. The first transition `Created -> Uploaded` can be implemented like this (pseudo code):

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== 'Created') {
    throw new Error('transition not allowed')
  }
  DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

The problem with this implementation is that it's not idempotent. Imagine that `SQS.sendMessage` fails. The state transition will fail, so you retry. But the second call to `uploaded(s3Key)` will throw a "transition not allowed" error because `DynamoDB.updateItem` was successful during the first call.

To fix that, you need to change the `if` statement to make the function idempotent:

```
uploaded(s3Key) {
  process = DynamoDB.getItem(processId)
  if (process.state !== 'Created' && process.state !== 'Uploaded') {
    throw new Error('transition not allowed')
  }
  DynamoDB.updateItem(processId, {'state': 'Uploaded', 'rawS3Key': s3Key})
  SQS.sendMessage({'processId': processId, 'action': 'process'});
}
```

If you retry now, you'll make multiple updates to Dynamo, which doesn't hurt. And you may send multiple SQS messages, which also doesn't hurt, because the SQS message consumer must be idempotent as well. The same applies to the transition `Uploaded -> Processed`.

Next, you'll begin to implement the Imagery server.

16.3.2 Implementing a fault-tolerant web service

We'll split the Imagery application into two parts: the web servers and the workers. As illustrated in figure 16.11, the web servers provide the REST API to the user, and the workers process images.

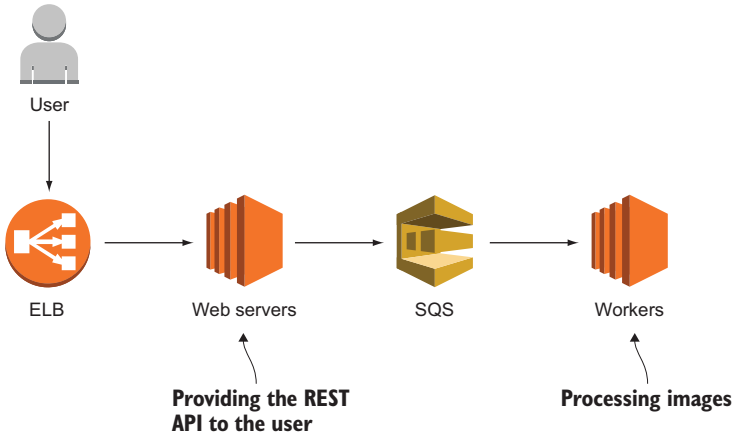


Figure 16.11 The Imagery application consists of two parts: the web servers and the workers.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. Imagery is located in /chapter16/.

The REST API will support the following routes:

- `POST /image`—A new image process is created when executing this route.
- `GET /image/:id`—This route returns the state of the process specified with the path parameter `:id`.
- `POST /image/:id/upload`—This route offers a file upload for the process specified with the `path` parameter `:id`.

To implement the web server, you'll again use Node.js and the Express web application framework. You'll use the Express framework, but don't feel intimidated, as you won't need to understand it in depth to follow along.

Setting up the web server project

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and things like that.

Listing 16.1 Initializing the Imagery server (server/server.js)

```
var express = require('express');          <-- Load Node.js modules (dependencies)
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var uuidv4 = require('uuid/v4');
var multiparty = require('multiparty');

var db = new AWS.DynamoDB({                <-- Creates a DynamoDB endpoint
```

```
  'region': 'us-east-1'
});
var sqs = new AWS.SQS({                    <-- Creates an SQS endpoint
  'region': 'us-east-1'
});
var s3 = new AWS.S3({                      <-- Creates an S3 endpoint
  'region': 'us-east-1'
});

var app = express();                       <-- Creates an Express application
app.use(bodyParser.json());                    Tells Express to parse the request bodies

// [...]

app.listen(process.env.PORT || 8080, function() {      <-- Starts Express on the port defined by the environment variable PORT, or defaults to 8080
  console.log('Server started. Open http://localhost:'
➥ + (process.env.PORT || 8080) + ' with browser.');
});
```

Don't worry too much about the boilerplate code; the interesting parts will follow.

CREATING A NEW IMAGERY PROCESS

To provide a REST API to create image processes, a fleet of EC2 instances will run Node.js code behind a load balancer. The image processes will be stored in DynamoDB. Figure 16.12 shows the flow of a request to create a new image process.

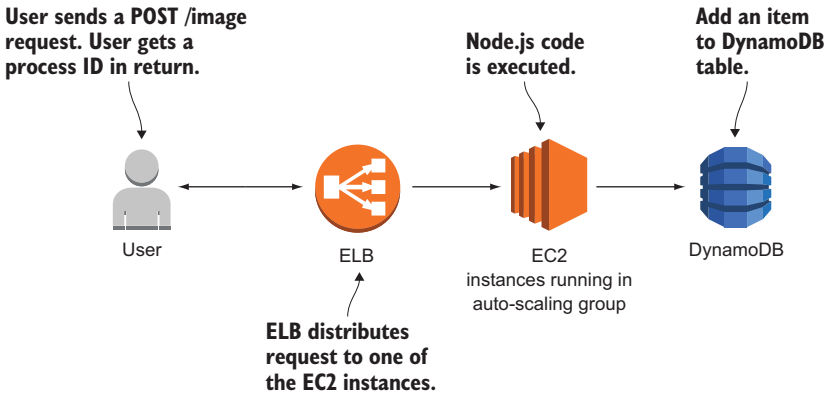
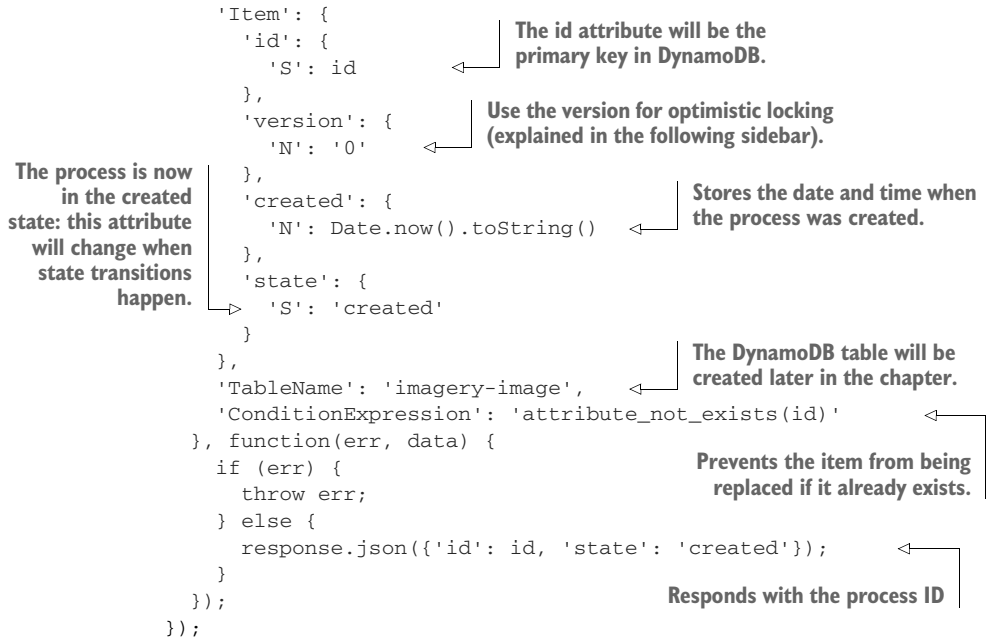


Figure 16.12 Creating a new image process in Imagery

You'll now add a route to the Express application to handle `POST /image` requests, as shown here.

Listing 16.2 Imagery server: `POST /image` creates an image process (server/server.js)

```
app.post('/image', function(request, response) {      <-- Registers the route with Express
    var id = uuidv4();                                <-- Creates a unique ID for the process
    db.putItem({   <-- Invokes the putItem operation on DynamoDB
```



A new process can now be created.

Optimistic locking

To prevent multiple updates to an DynamoDB item, you can use a trick called *optimistic locking*. When you want to update an item, you must specify which version you want to update. If that version doesn't match the current version of the item in the database, your update will be rejected. Keep in mind that optimistic locking is your responsibility, not a default available in DynamoDB. DynamoDB only provides the features to implement optimistic locking.

Imagine the following scenario. An item is created in version 0. Process A looks up that item (version 0). Process B also looks up that item (version 0). Now process A wants to make a change by invoking the `updateItem` operation on DynamoDB. Therefore process A specifies that the expected version is 0. DynamoDB will allow that modification, because the version matches; but DynamoDB will also change the item's version to 1 because an update was performed. Now process B wants to make a modification and sends a request to DynamoDB with the expected item version 0. DynamoDB will reject that modification because the expected version doesn't match the version DynamoDB knows of, which is 1.

To solve the problem for process B, you can use the same trick introduced earlier: retry. Process B will again look up the item, now in version 1, and can (you hope) make the change.

(continued)

There is one problem with optimistic locking: If many modifications happen in parallel, a lot of overhead is created because of many retries. But this is only a problem if you expect a lot of concurrent writes to a single item, which can be solved by changing the data model. That's not the case in the Imagery application. Only a few writes are expected to happen for a single item: optimistic locking is a perfect fit to make sure you don't have two writes where one overrides changes made by another.

The opposite of *optimistic locking* is *pessimistic locking*. A pessimistic lock strategy can be implemented by using a semaphore. Before you change data, you need to lock the semaphore. If the semaphore is already locked, you wait until the semaphore becomes free again.

The next route you need to implement is to look up the current state of a process.

LOOKING UP AN IMAGERY PROCESS

You'll now add a route to the Express application to handle `GET /image/:id` requests. Figure 16.13 shows the request flow.

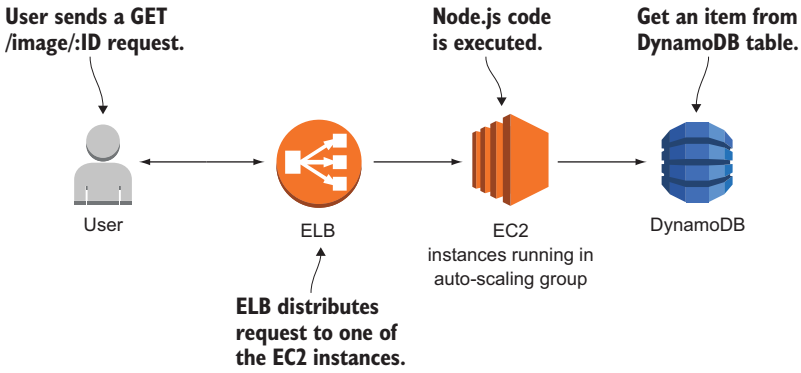


Figure 16.13 Looking up an image process in Imagery to return its state

Express will take care of the path parameter `:id`; Express will provide it within `request.params.id`. The implementation needs to get an item from DynamoDB based on the path parameter ID.

Listing 16.3 `GET /image/:id` looks up an image process (server/server.js)

```
function mapImage = function(item) {
  return {
    'id': item.id.S,
    'version': parseInt(item.version.N, 10),
    'state': item.state.S,
    'rawS3Key': // [...]
    'processedS3Key': // [...]
```

Helper function to map a DynamoDB result to a JavaScript object

```
    'processedImage': // [...]
  };
};

function getImage(id, cb) {
  db.getItem({                          <-- Invokes the getItem operation on DynamoDB
    'Key': {
      'id': {                           <-- id is the partition key.
        'S': id
      }
    },
    'TableName': 'imagery-image'
  }, function(err, data) {
    if (err) {
      cb(err);
    } else {
      if (data.Item) {
        cb(null, lib.mapImage(data.Item));
      } else {
        cb(new Error('image not found'));
      }
    }
  });
}

app.get('/image/:id', function(request, response) {   <-- Registers the route with Express
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      response.json(image);             <-- Responds with the image process
    }
  });
});
```

The only thing missing is the upload part, which comes next.

Uploading an image

Uploading an image via `POST` request requires several steps:

1. Upload the raw image to S3.
2. Modify the item in DynamoDB.
3. Send an SQS message to trigger processing.

Figure 16.14 shows this flow.

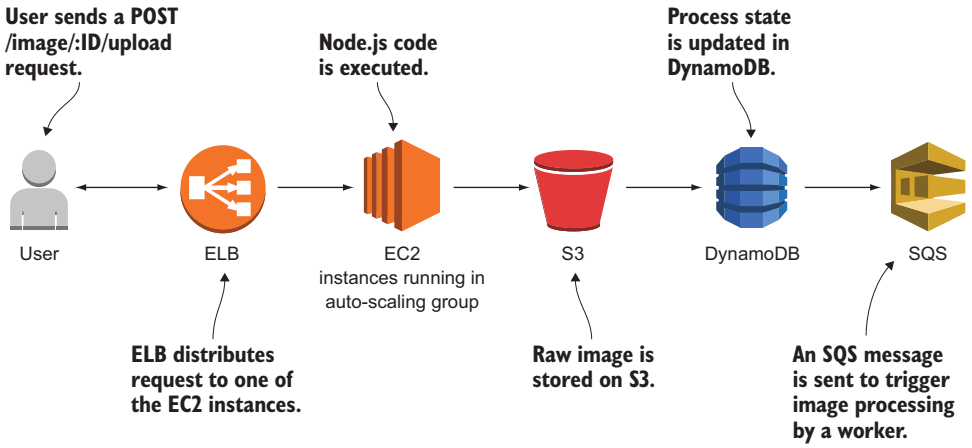


Figure 16.14 Uploading a raw image to Imagery and trigger image processing

Ths listing shows the implementation of these steps.

Listing 16.4 `POST /image/:id/upload` uploads an image (server/server.js)

```
function uploadImage(image, part, response) {
  var rawS3Key = 'upload/' + image.id + '-' + Date.now();
  s3.putObject({
    'Bucket': process.env.ImageBucket,
    'Key': rawS3Key,
    'Body': part,
    'ContentLength': part.byteCount
  }, function(err, data) {
    if (err) {
      throw err;
    } else {
      db.updateItem({
        'Key': {
          'id': {
            'S': image.id
          }
        },
        'UpdateExpression': 'SET #s=:newState,
➥ version=:newVersion, rawS3Key=:rawS3Key',
        'ConditionExpression': 'attribute_exists(id)
➥ AND version=:oldVersion
➥ AND #s IN (:stateCreated, :stateUploaded)',
        'ExpressionAttributeNames': {
          '#s': 'state'
        },
        'ExpressionAttributeValues': {
          ':newState': {
            'S': 'uploaded'
          },
          ':oldVersion': {
```

Creates a key for the S3 object

Calls the S3 API to upload an object

The S3 bucket name is passed in as an environment variable (the bucket will be created later in the chapter).

The body is the uploaded stream of data.

Calls the DynamoDB API to update an object.

Updates the state, version, and raw S3 key

Updates only when item exists. Version equals the expected version, and state is one of those allowed.

```
            'N': image.version.toString()
          },
          ':newVersion': {
            'N': (image.version + 1).toString()
          },
          ':rawS3Key': {
            'S': rawS3Key
          },
          ':stateCreated': {
            'S': 'created'
          },
          ':stateUploaded': {
            'S': 'uploaded'
          }
        },
        'ReturnValues': 'ALL_NEW',
        'TableName': 'imagery-image'
      }, function(err, data) {
        if (err) {
          throw err;
        } else {
          sqs.sendMessage({
            'MessageBody': JSON.stringify(
              {'imageId': image.id, 'desiredState': 'processed'}
            ),
            'QueueUrl': process.env.ImageQueue,
          }, function(err) {
            if (err) {
              throw err;
            } else {
              response.redirect('/#view=' + image.id);
              response.end();
            }
          });
        }
      });
    }
  });
}

app.post('/image/:id/upload', function(request, response) {
  getImage(request.params.id, function(err, image) {
    if (err) {
      throw err;
    } else {
      var form = new multiparty.Form();
      form.on('part', function(part) {
        uploadImage(image, part, response);
      });
      form.parse(request);
    }
  });
});
```

Calls the SQS API to publish a message

Creates the message body containing the image's ID and the desired state

The queue URL is passed in as an environment variable.

Registers the route with Express

We are using the multiparty module to handle multi-part uploads.

The server side is finished. Next you'll continue to implement the processing part in the Imagery worker. After that, you can deploy the application.

16.3.3 Implementing a fault-tolerant worker to consume SQS messages

The Imagery worker does the asynchronous stuff in the background: processing images by applying a sepia filter. The worker handles consuming SQS messages and processing images. Fortunately, consuming SQS messages is a common task that is solved by Elastic Beanstalk, which you'll use later to deploy the application. Elastic Beanstalk can be configured to listen to SQS messages and execute an HTTP `POST` request for every message. In the end, the worker implements a REST API that is invoked by Elastic Beanstalk. To implement the worker, you'll again use Node.js and the Express framework.

SETTING UP THE SERVER PROJECT

As always, you need some boilerplate code to load dependencies, initial AWS endpoints, and so on.

Listing 16.5 Initializing the Imagery worker (worker/worker.js)

```
var express = require('express');
var bodyParser = require('body-parser');
var AWS = require('aws-sdk');
var assert = require('assert-plus');
var Caman = require('caman').Caman;
var fs = require('fs');

var db = new AWS.DynamoDB({
  'region': 'us-east-1'
});
var s3 = new AWS.S3({
  'region': 'us-east-1'
});

var app = express();
app.use(bodyParser.json());

app.get('/', function(request, response) {
  response.json({});
});

// [...]

app.listen(process.env.PORT || 8080, function() {
  console.log('Worker started on port ' + (process.env.PORT || 8080));
});
```

Loads Node.js modules (dependencies)

Creates a DynamoDB endpoint

Creates an S3 endpoint

Creates an Express application

Registers a route for health checks that returns an empty object

Starts Express on a port defined by the environment variable PORT, or defaults to 8080

The Node.js module `caman` is used to create sepia images. You'll wire that up next.

HANDLING SQS MESSAGES AND PROCESSING THE IMAGE

The SQS message to trigger the image processing is handled in the worker. Once a message is received, the worker starts to download the raw image from S3, applies the sepia filter, and uploads the processed image back to S3. After that, the process state in DynamoDB is modified. Figure 16.15 shows the steps.

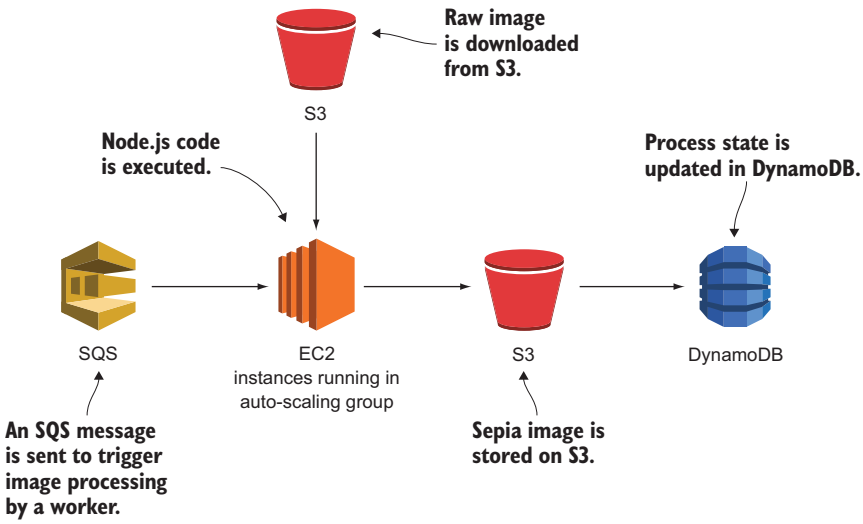


Figure 16.15 Processing a raw image to upload a sepia image to S3

Instead of receiving messages directly from SQS, you'll take a shortcut. Elastic Beanstalk—the deployment tool you'll use—provides a feature that consumes messages from a queue and invokes a HTTP `POST` request for every message. You configure the `POST` request to be made to the resource `/sqs`.

Listing 16.6 Imagery worker: `POST /sqs` handles SQS messages (worker/worker.js)

```
function processImage(image, cb) {
  var processedS3Key = 'processed/' + image.id + '-' + Date.now() + '.png';
  // download raw image from S3
  // process image
  // upload sepia image to S3
  cb(null, processedS3Key);
}

function processed(image, request, response) {
  processImage(image, function(err, processedS3Key) {
    if (err) {
      throw err;
    } else {
      db.updateItem({
```

The implementation of processImage isn't shown here; you can find it in the book's source code folder.

Invokes the updateItem operation on DynamoDB

```
        'Key': {
          'id': {
            'S': image.id
          }
        },
        'UpdateExpression': 'SET #s=:newState,
➥ version=:newVersion, processedS3Key=:processedS3Key',
        'ConditionExpression': 'attribute_exists(id)
➥ AND version=:oldVersion
➥ AND #s IN (:stateUploaded, :stateProcessed)',
        'ExpressionAttributeNames': {
          '#s': 'state'
        },
        'ExpressionAttributeValues': {
          ':newState': {
            'S': 'processed'
          },
          ':oldVersion': {
            'N': image.version.toString()
          },
          ':newVersion': {
            'N': (image.version + 1).toString()
          },
          ':processedS3Key': {
            'S': processedS3Key
          },
          ':stateUploaded': {
            'S': 'uploaded'
          },
          ':stateProcessed': {
            'S': 'processed'
          }
        },
        'ReturnValues': 'ALL_NEW',
        'TableName': 'imagery-image'
      }, function(err, data) {
        if (err) {
          throw err;
        } else {
          response.json(lib.mapImage(data.Attributes));
        }
      });
    }
  });
}

app.post('/sqs', function(request, response) {
  assert.string(request.body.imageId, 'imageId');
  assert.string(request.body.desiredState, 'desiredState');
  getImage(request.body.imageId, function(err, image) {

    if (err) {
      throw err;
    } else {
      if (request.body.desiredState === 'processed') {
```

Updates the state, version, and processed S3 key

Updates only when an item exists, version equals the expected version, and state is one of those allowed.

Responds with the process's new state

Registers the route with Express

The implementation of getImage is the same as on the server.

```
        processed(image, request, response);
      } else {
        throw new Error("unsupported desiredState");
      }
    }
  });
});
```

Invokes the processed function if the SQS message's desiredState equals "processed".

If the `POST /sqs` route responds with a 2XX HTTP status code, Elastic Beanstalk considered the message delivery successful and deletes the message from the queue. Otherwise, the message is redelivered.

Now you can process the SQS message to transform the raw image and upload the sepia version to S3. The next step is to deploy all that code to AWS in a fault-tolerant way.

16.3.4 Deploying the application

As mentioned previously, you'll use Elastic Beanstalk to deploy the server and the worker. You'll use CloudFormation to do so. This may sound strange, because you're using an automation tool to use another automation tool. But CloudFormation does a bit more than just deploying two Elastic Beanstalk applications. It defines the following:

- The S3 bucket for raw and processed images
- The DynamoDB table `imagery-image`
- The SQS queue and dead-letter queue
- IAM roles for the server and worker EC2 instances
- Elastic Beanstalk applications for the server and worker

It takes quite a while to create that CloudFormation stack; that's why you should do so now. After you've created the stack, we'll look at the template. After that, the stack should be ready to use.

To help you deploy Imagery, we created a CloudFormation template located at http://mng.bz/Z33C. Create a stack based on that template. The stack output `EndpointURL` returns the URL that you can access from your browser to use Imagery. Here's how to create the stack from the terminal:

```
$ aws cloudformation create-stack --stack-name imagery \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter16/template.yaml \
➥ --capabilities CAPABILITY_IAM
```

Now let's look at the CloudFormation template.

Deploying S3, DynamoDB, and SQS

Listing 16.7 describes the S3 bucket, DynamoDB table, and SQS queue.

Listing 16.7 Imagery CloudFormation template: S3, DynamoDB, and SQS

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 16'
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
Resources:
  Bucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: !Sub 'imagery-${AWS::AccountId}'
      WebsiteConfiguration:
        ErrorDocument: error.html
        IndexDocument: index.html
  Table:
    Type: 'AWS::DynamoDB::Table'
    Properties:
      AttributeDefinitions:
      - AttributeName: id
        AttributeType: S
      KeySchema:
      - AttributeName: id
        KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 1
        WriteCapacityUnits: 1
      TableName: 'imagery-image'
  SQSDLQueue:
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: 'message-dlq'
  SQSQueue:
    Type: 'AWS::SQS::Queue'
    Properties:
      QueueName: message
      RedrivePolicy:
        deadLetterTargetArn: !Sub '${SQSDLQueue.Arn}'
        maxReceiveCount: 10
# [...]
Outputs:
  EndpointURL:
    Value: !Sub 'http://${EBServerEnvironment.EndpointURL}'
    Description: Load Balancer URL
```

S3 bucket for uploaded and processed images, with web hosting enabled

The bucket name contains the account ID to make the name unique.

DynamoDB table containing the image processes

The id attribute is used as the partition key.

SQS queue that receives messages that can't be processed

SQS queue to trigger image processing

If a message is received more than 10 times, it's moved to the dead-letter queue.

Visit the output with your browser to use Imagery.

The concept of a *dead-letter queue* (DLQ) needs a short introduction here as well. If a single SQS message can't be processed, the message becomes visible again on the queue for other workers. This is called a *retry*. But if for some reason every retry fails (maybe you have a bug in your code), the message will reside in the queue forever and may waste a lot of resources because of all the retries. To avoid this, you can configure a dead-letter

queue. If a message is retried more than a specific number of times, it's removed from the original queue and forwarded to the DLQ. The difference is that no worker listens for messages on the DLQ. But you should create a CloudWatch alarm that triggers if the DLQ contains more than zero messages, because you need to investigate this problem manually by looking at the message in the DLQ.

Now that the basic resources have been designed, let's move on to the more specific resources.

IAM roles for server and worker EC2 instances

Remember that it's important to only grant the privileges that are necessary. All server instances must be able to do the following:

- `sqs:SendMessage` to the SQS queue created in the template to trigger image processing
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the `upload/` key prefix)
- `dynamodb:GetItem`, `dynamodb:PutItem`, and `dynamodb:UpdateItem` to the DynamoDB table created in the template
- `cloudwatch:PutMetricData`, which is an Elastic Beanstalk requirement
- `s3:Get*`, `s3:List*`, and `s3:PutObject`, which is an Elastic Beanstalk requirement

All worker instances must be able to do the following:

- `sqs:ChangeMessageVisibility`, `sqs:DeleteMessage`, and `sqs:ReceiveMessage` to the SQS queue created in the template
- `s3:PutObject` to the S3 bucket created in the template to upload a file to S3 (you can further limit writes to the `processed/` key prefix)
- `dynamodb:GetItem` and `dynamodb:UpdateItem` to the DynamoDB table created in the template
- `cloudwatch:PutMetricData`, which is an Elastic Beanstalk requirement
- `s3:Get*`, `s3:List*`, and `s3:PutObject`, which is an Elastic Beanstalk requirement

If you don't feel comfortable with IAM roles, take a look at the book's code repository on GitHub at https://github.com/AWSinAction/code2. The template with IAM roles can be found in /chapter16/template.yaml.

Now it's time to design the Elastic Beanstalk applications.

Elastic Beanstalk for the server

First off, a short refresher on Elastic Beanstalk, which we touched on in chapter 5. Elastic Beanstalk consists of these elements:

- An *application* is a logical container. It contains versions, environments, and configurations. To use AWS Elastic Beanstalk in a region, you have to create an application first.
- A *version* contains a specific release of your application. To create a new version, you have to upload your executables (packed into an archive) to S3. A version is basically a pointer to this archive of executables.

- A *configuration template* contains your default configuration. You can manage the configuration of your application (such as the port your application listens on) as well as the configuration of the environment (such as the size of the virtual server) with your custom configuration template.
- An *environment* is the place where AWS Elastic Beanstalk executes your application. It consists of a version and the configuration. You can run multiple environments for one application by using the versions and configurations multiple times.

Figure 16.16 shows the parts of an Elastic Beanstalk application.

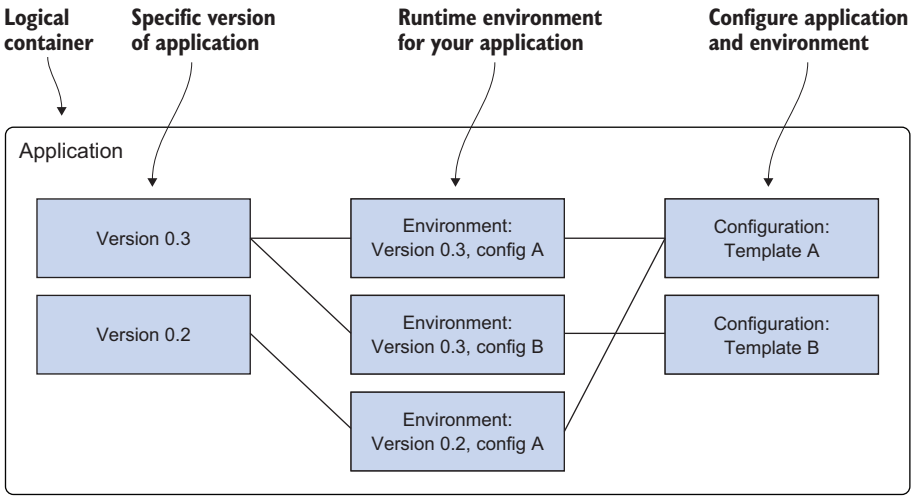


Figure 16.16 An AWS Elastic Beanstalk application consists of versions, environments, and configurations.

Now that you've refreshed your memory, let's look at the Elastic Beanstalk application that deploys the Imagery server.

Listing 16.8 Imagery CloudFormation template: Elastic Beanstalk for the server

```
EBServerApplication:                                    Describes the server
  Type: 'AWS::ElasticBeanstalk::Application'            application container
  Properties:
    ApplicationName: 'imagery-server'
    Description: 'Imagery server: AWS in Action: chapter 16'
EBServerConfigurationTemplate:
  Type: 'AWS::ElasticBeanstalk::ConfigurationTemplate'
  Properties:
    ApplicationName: !Ref EBServerApplication
    Description: 'Imagery server: AWS in Action: chapter 16'
    SolutionStackName:                                  Uses Amazon Linux 2017.09
➥ '64bit Amazon Linux 2017.09 v4.4.0 running Node.js'  running Node.js 6.11.5 per default
    OptionSettings:
    - Namespace: 'aws:autoscaling:asg'                  Minimum of two EC2 instances for
      OptionName: 'MinSize'                             fault-tolerance
      Value: '2'                            )
```

```
      - Namespace: 'aws:autoscaling:launchconfiguration'
        OptionName: 'EC2KeyName'
        Value: !Ref KeyName
      - Namespace: 'aws:autoscaling:launchconfiguration'
        OptionName: 'IamInstanceProfile'
        Value: !Ref ServerInstanceProfile
      - Namespace: 'aws:elasticbeanstalk:container:nodejs'
        OptionName: 'NodeCommand'
        Value: 'node server.js'
      - Namespace: 'aws:elasticbeanstalk:application:environment'
        OptionName: 'ImageQueue'
        Value: !Ref SQSQueue
      - Namespace: 'aws:elasticbeanstalk:application:environment'
        OptionName: 'ImageBucket'
        Value: !Ref Bucket
      - Namespace: 'aws:elasticbeanstalk:container:nodejs:staticfiles'
        OptionName: '/public'
        Value: '/public'
EBServerApplicationVersion:
  Type: 'AWS::ElasticBeanstalk::ApplicationVersion'
  Properties:
    ApplicationName: !Ref EBServerApplication
    Description: 'Imagery server: AWS in Action: chapter 16'
    SourceBundle:
      S3Bucket: 'awsinaction-code2'
      S3Key: 'chapter16/build/server.zip'
EBServerEnvironment:
  Type: 'AWS::ElasticBeanstalk::Environment'
  Properties:
    ApplicationName: !Ref EBServerApplication
    Description: 'Imagery server: AWS in Action: chapter 16'
    TemplateName: !Ref EBServerConfigurationTemplate
    VersionLabel: !Ref EBServerApplicationVersion
```

Passes a value from the KeyName parameter

Links to the IAM instance profile created in the previous section

Start command

Passes the SQS queue into an environment variable

Passes the S3 bucket into an environment variable

Serves all files from /public as static files

Loads code from the book's S3 bucket

Under the hood, Elastic Beanstalk uses an ELB to distribute the traffic to the EC2 instances that are also managed by Elastic Beanstalk. You only need to worry about the configuration of Elastic Beanstalk and the code.

Elastic Beanstalk for the worker

The worker Elastic Beanstalk application is similar to the server. The differences are annotated in the following listing.

Listing 16.9 Imagery CloudFormation template: Elastic Beanstalk for the worker

```
EBWorkerApplication:
  Type: 'AWS::ElasticBeanstalk::Application'
  Properties:
    ApplicationName: 'imagery-worker'
    Description: 'Imagery worker: AWS in Action: chapter 16'
EBWorkerConfigurationTemplate:
  Type: 'AWS::ElasticBeanstalk::ConfigurationTemplate'
  Properties:
    ApplicationName: !Ref EBWorkerApplication
```

Describes the worker application container

```
    Description: 'Imagery worker: AWS in Action: chapter 16'
    SolutionStackName:
➥ '64bit Amazon Linux 2017.09 v4.4.0 running Node.js'
    OptionSettings:
    - Namespace: 'aws:autoscaling:launchconfiguration'
      OptionName: 'EC2KeyName'
      Value: !Ref KeyName
    - Namespace: 'aws:autoscaling:launchconfiguration'
      OptionName: 'IamInstanceProfile'
      Value: !Ref WorkerInstanceProfile
    - Namespace: 'aws:elasticbeanstalk:sqsd'
      OptionName: 'WorkerQueueURL'
      Value: !Ref SQSQueue
    - Namespace: 'aws:elasticbeanstalk:sqsd'
      OptionName: 'HttpPath'
      Value: '/sqs'
    - Namespace: 'aws:elasticbeanstalk:container:nodejs'
      OptionName: 'NodeCommand'
      Value: 'node worker.js'
    - Namespace: 'aws:elasticbeanstalk:application:environment'
      OptionName: 'ImageQueue'
      Value: !Ref SQSQueue
    - Namespace: 'aws:elasticbeanstalk:application:environment'
      OptionName: 'ImageBucket'
      Value: !Ref Bucket
EBWorkerApplicationVersion:
  Type: 'AWS::ElasticBeanstalk::ApplicationVersion'
  Properties:
    ApplicationName: !Ref EBWorkerApplication
    Description: 'Imagery worker: AWS in Action: chapter 16'
    SourceBundle:
      S3Bucket: 'awsinaction-code2'
      S3Key: 'chapter16/build/worker.zip'
EBWorkerEnvironment:
  Type: 'AWS::ElasticBeanstalk::Environment'
  Properties:
    ApplicationName: !Ref EBWorkerApplication
    Description: 'Imagery worker: AWS in Action: chapter 16'
    TemplateName: !Ref EBWorkerConfigurationTemplate
    VersionLabel: !Ref EBWorkerApplicationVersion
    Tier:
      Type: 'SQS/HTTP'
      Name: 'Worker'
      Version: '1.0'
```

Configures the HTTP resource that is invoked when an SQS message is received

Switches to the worker environment tier (pushes SQS messages to your app)

After all that YAML reading, the CloudFormation stack should be created. Verify the status of your stack:

```
$ aws cloudformation describe-stacks --stack-name imagery
{
  "Stacks": [{
    [...]
    "Description": "AWS in Action: chapter 16",
    "Outputs": [{
```

```
      "Description": "Load Balancer URL",
      "OutputKey": "EndpointURL",
      "OutputValue": "http://awseb-...582.us-east-1.elb.amazonaws.com"
    }],
    "StackName": "imagery",
    "StackStatus": "CREATE_COMPLETE"
  }]
}
```

Copy this output into your web browser.

Wait until CREATE_COMPLETE is reached.

The `EndpointURL` output of the stack contains the URL to access the Imagery application. When you open Imagery in your web browser, you can upload an image as shown in figure 16.17.

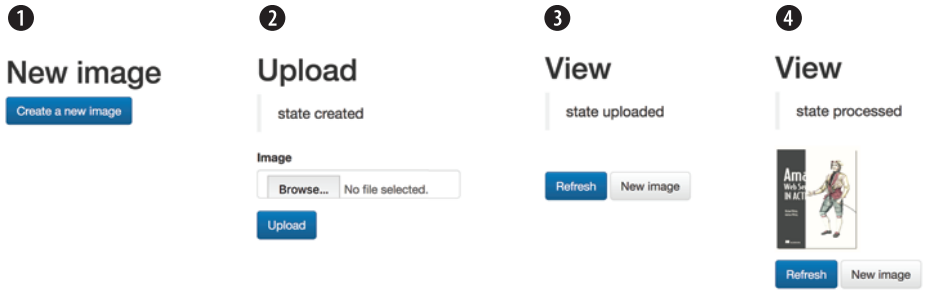


Figure 16.17 The Imagery application in action

Go ahead and upload some images and enjoy watching the images being processed.

Cleaning up

To get the name of the S3 bucket used by Imagery, run the following command in your terminal.

```
$ aws cloudformation describe-stack-resource --stack-name imagery \
➥ --logical-resource-id Bucket \
➥ --query "StackResourceDetail.PhysicalResourceId"
➥ --output text
imagery-000000000000
```

Delete all the files in your S3 bucket `imagery-000000000000`. Don't forget to replace *$bucketname* with the output from the previous command.

```
$ aws s3 rm s3://$bucketname --recursive
```

Execute the following command to delete the CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name imagery
```

Stack deletion will take some time.

Congratulations, you have accomplished a big milestone: building a fault tolerant application on AWS. You are only one step away from the end game, which is scaling your application dynamically based on load.

Summary

- Fault tolerance means expecting that failures happen, and designing your systems in such a way that they can deal with failure.
- To create a fault-tolerant application, you can use idempotent actions to transfer from one state to the next.
- State shouldn't reside on the EC2 instance (a stateless server) as a prerequisite for fault-tolerance.
- AWS offers fault-tolerant services and gives you all the tools you need to create fault-tolerant systems. EC2 is one of the few services that isn't fault-tolerant out of the box.
- You can use multiple EC2 instances to eliminate the single point of failure. Redundant EC2 instances in different availability zones, started with an auto-scaling group, are the way to make EC2 fault-tolerant.

17 Scaling up and down: auto-scaling and CloudWatch

This chapter covers

- Creating an auto-scaling group with launch configuration
- Using auto-scaling to change the number of virtual machines
- Scaling a synchronous decoupled app behind a load balancer (ALB)
- Scaling an asynchronous decoupled app using a queue (SQS)

Suppose you're organizing a party to celebrate your birthday. How much food and drink do you need to buy? Calculating the right numbers for your shopping list is difficult:

- *How many people will attend?* You received several confirmations, but some guests will cancel at short notice or show up without letting you know in advance. So the number of guests is vague.

- *How much will your guests eat and drink?* Will it be a hot day, with everybody drinking a lot? Will your guests be hungry? You need to guess the demand for food and drink based on experiences from previous parties.

Solving the equation is a challenge because there are many unknowns. Behaving as a good host, you'll order more food and drink as needed, and no guest will be hungry or thirsty for long.

Planning to meet future demands is nearly impossible. To prevent a supply gap, you need to add extra capacity on top of the planned demand to prevent running short of resources.

Before the cloud, the same was true for our industry when planning the capacity of our IT infrastructure. When procuring hardware for a data center, we always had to buy hardware based on the demands of the future. There were many uncertainties when making these decisions:

- How many users needed to be served by the infrastructure?
- How much storage would the users need?
- How much computing power would be required to handle their requests?

To avoid supply gaps, we had to order more or faster hardware than needed, causing unnecessary expenses. On AWS, you can use services on demand. Planning capacity is less and less important. You can scale from one EC2 instance to thousands of EC2 instances. Storage can grow from gigabytes to petabytes. You can scale on demand, thus replacing capacity planning. The ability to scale on demand is called *elasticity* by AWS.

Public cloud providers like AWS can offer the needed capacity with a short waiting time. AWS serves millions of customers, and at that scale it isn't a problem to provide you with 100 additional virtual machines within minutes if you need them suddenly. This allows you to address another problem: typical traffic patterns, as shown in figure 17.1. Think about the load on your infrastructure during the day versus at night, on a weekday versus the weekend, or before Christmas versus the rest of year. Wouldn't it be nice if you could add capacity when traffic grows and remove capacity when traffic shrinks? In this chapter, you'll learn how to scale the number of virtual machines based on current load.

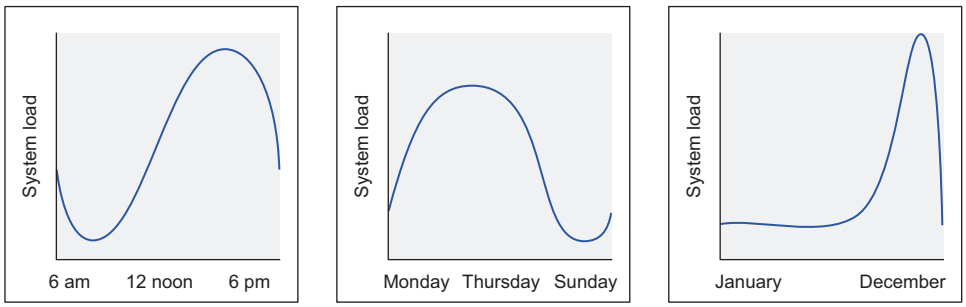


Figure 17.1 Typical traffic patterns for a web shop

Scaling the number of virtual machines is possible with auto-scaling groups (ASG) and scaling policies on AWS. Auto-scaling is part of the EC2 service and helps you to scale the number of EC2 instances you need to fulfill the current load of your system. We introduced auto-scaling groups in chapter 14 to ensure that a single virtual machine was running even if an outage of an entire data center occurred. In this chapter, you'll learn how to use a dynamic EC2 instance pool:

- Using auto-scaling groups to launch multiple virtual machines of the same kind.
- Changing the number of virtual machines based on CPU load with the help of CloudWatch alarms.
- Changing the number of virtual machines based on a schedule, to adapt to recurring traffic patterns.
- Using a load balancer as an entry point to the dynamic EC2 instance pool.
- Using a queue to decouple the jobs from the dynamic EC2 instance pool.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. As long as you don't run the examples longer than a few days, you won't pay anything for it. Keep in mind that this applies only if you created a fresh AWS account for this book and there is nothing else going on in your AWS account. Try to complete the chapter within a few days, because you'll clean up your account at the end of the chapter.

There are two prerequisites for being able to scale your application horizontally, which means increasing and decreasing the number of virtual machines based on the current workload:

- *The EC2 instances you want to scale need to be stateless.* You can achieve stateless servers by storing data with the help of service like RDS (SQL database), DynamoDB (NoSQL database), EFS (network filesystem), or S3 (object store) instead of storing data on disks (instance store or EBS) that are only available to a single EC2 instance.
- *An entry point to the dynamic EC2 instance pool is needed to be able to distribute the workload across multiple EC2 instances.* EC2 instances can be decoupled synchronously with a load balancer, or asynchronously with a queue.

We introduced the concept of the stateless server in part 3 of this book and explained how to use decoupling in chapter 15. In this chapter you'll return to the concept of the stateless server and also work through an example of synchronous and asynchronous decoupling.

17.1 Managing a dynamic EC2 instance pool

Imagine that you need to provide a scalable infrastructure to run a web application, such as a blogging platform. You need to launch uniform virtual machines when the number of requests grows, and terminate virtual machines when the number of requests shrinks. To adapt to the current workload in an automated way, you need to

be able to launch and terminate VMs automatically. Therefore, the configuration and deployment of the web application needs to be done during bootstrapping, without human interaction.

AWS offers a service to manage such a dynamic EC2 instance pool, called *auto-scaling groups*. Auto-scaling groups help you to:

- Dynamically adjust the number of virtual machines that are running
- Launch, configure, and deploy uniform virtual machines

The auto-scaling group grows and shrinks within the bounds you define. Defining a minimum of two virtual machines allows you to make sure at least two virtual machines are running in different availability zones to plan for failure. Conversely, defining a maximum number of virtual machines ensures you are not spending more money than you intended for your infrastructure.

As figure 17.2 shows, auto-scaling consists of three parts:

1. A launch configuration that defines the size, image, and configuration of virtual machines.
2. An auto-scaling group that specifies how many virtual machines need to be running based on the launch configuration.
3. Scaling plans that adjust the desired number of EC2 instances in the auto-scaling group based on a plan or dynamically.

Because the auto-scaling group requires a launch configuration, you need to create one before you can create an auto-scaling group. If you use a template, as you will in this chapter, this dependency will be resolved by CloudFormation automatically.

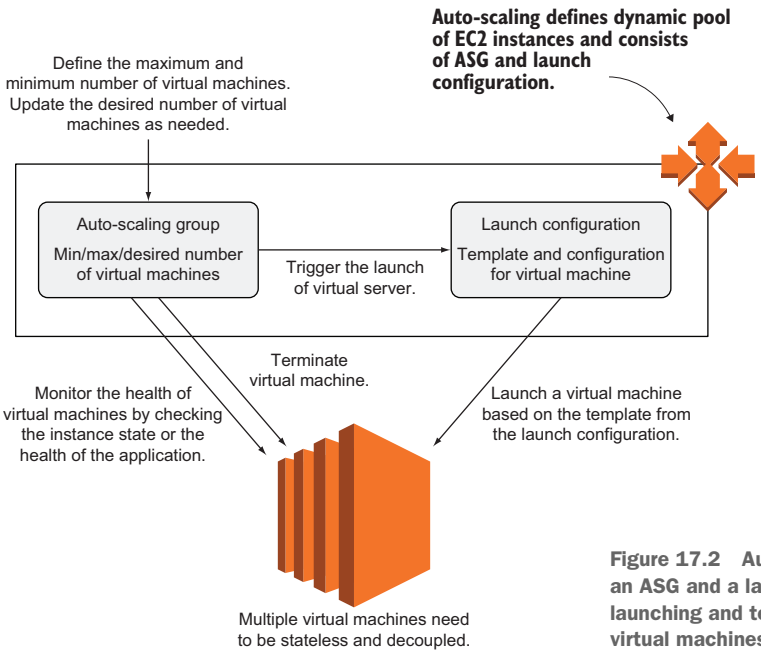


Figure 17.2 Auto-scaling consists of an ASG and a launch configuration, launching and terminating uniform virtual machines.

If you want multiple EC2 instances to handle a workload, it's important to start identical virtual machines to build a homogeneous foundation. You use a launch configuration to define and configure new virtual machines. Table 17.1 shows the most important parameters for a launch configuration.

Table 17.1 Launch configuration parameters

Name	Description	Possible values
`ImageId`	Image from which to start a virtual machine	ID of Amazon Machine Image (AMI)
`InstanceType`	Size for new virtual machines	Instance type (such as t2.micro)
`UserData`	User data for the virtual machine used to execute a script during bootstrapping	BASE64-encoded String
`KeyName`	The key pair used to authenticate via SSH	Name of an EC2 key pair
`AssociatePublicIpAddress`	Associates a public IP address to the virtual machine	True or false
`SecurityGroups`	Attaches security groups to new virtual machines	List of security group names
`IamInstanceProfile`	Attaches an IAM instance profile linked to an IAM role	Name or Amazon Resource Name (ARN, an ID) of an IAM instance profile

After you create a launch configuration, you can create an auto-scaling group that references it. The auto-scaling group defines the maximum, minimum, and desired number of virtual machines. *Desired* means this number of EC2 instances should be running. If the current number of EC2 instances is below the desired number, the auto-scaling group will add EC2 instances. If the current number of EC2 instances is above the desired number, EC2 instances will be removed. The desired capacity can be changed automatically based on load or a schedule or manually. *Minimum* and *maximum* are the lower and upper limit for the number of virtual machines within the auto-scaling group.

The auto-scaling group also monitors whether EC2 instances are healthy and replaces broken instances. Table 17.2 shows the most important parameters for an auto-scaling group.

Table 17.2 Auto-scaling group (ASG) parameters

Name	Description	Possible values
`DesiredCapacity`	Desired number of healthy virtual machines	Integer
`MaxSize`	Maximum number of virtual machines, the upper scaling limit	Integer

Table 17.2 Auto-scaling group (ASG) parameters *(continued)*

Name	Description	Possible values
`MinSize`	Minimum number of virtual machines, the lower scaling limit	Integer
`HealthCheckType`	How the auto-scaling group checks the health of virtual machines	`EC2` (health of the instance) or `ELB` (health check of instance performed by a load balancer)
`HealthCheckGracePeriod`	Period for which the health check is paused after the launch of a new instance, to wait until the instance is fully bootstrapped	Number of seconds
`LaunchConfigurationName`	Name of the launch configuration used to start new virtual machines	Name of a launch configuration
`TargetGroupARNs`	The target groups of a load balancer, where auto-scaling registers new instances automatically	List of target group ARNs
`VPCZoneIdentifier`	List of subnets in which to launch EC2 instances in	List of subnet identifiers of a VPC

If you specify multiple subnets with the help of `VPCZoneIdentifier` for the auto-scaling group, EC2 instances will be evenly distributed among these subnets and thus among availability zones.

Don't forget to define health check grace period

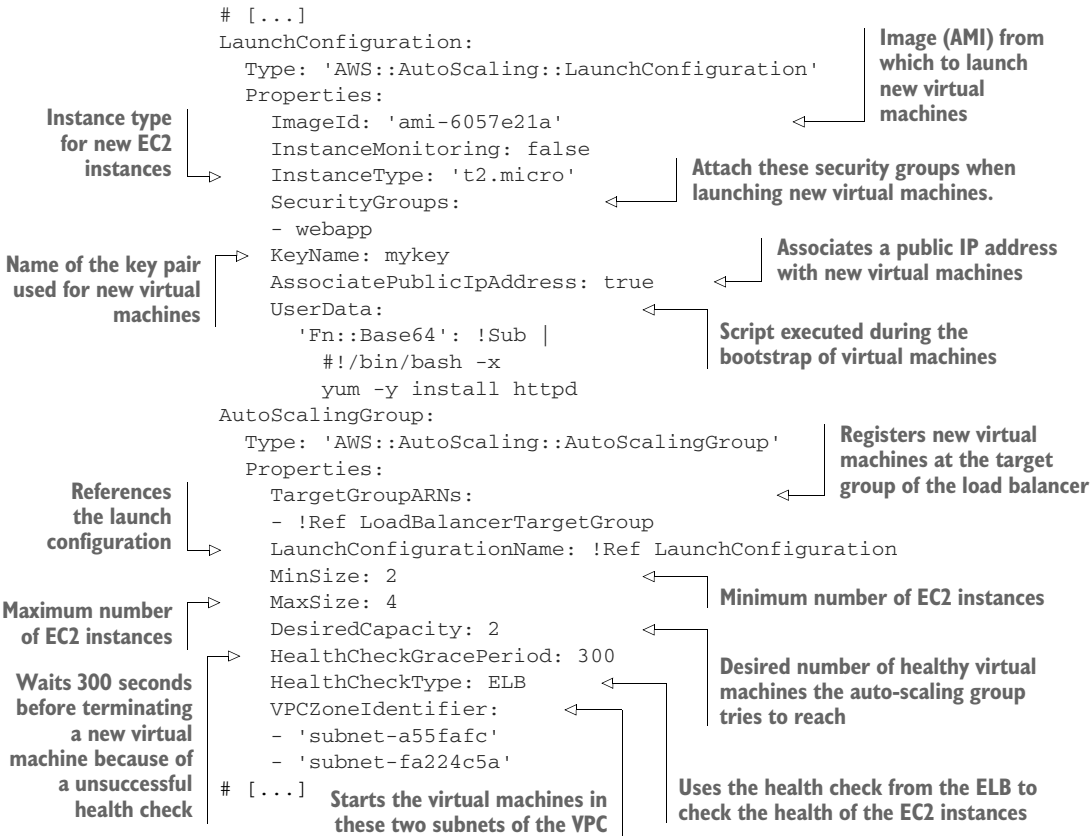
If you are using the ELB's health check for your auto-scaling group, make sure you specify a `HealthCheckGracePeriod` as well. Specify a health check grace period based on the time it takes from launching an EC2 instance until your application is running and passes the ELB's health check. For a simple web application, a health check period of 5 minutes is suitable.

You can't edit a launch configuration, for example to change the instance type, the machine image (AMI), or the security groups of your instances. If you need to make changes to a launch configuration, follow these steps:

1. Create a launch configuration.
2. Edit the auto-scaling group, and reference the new launch configuration.
3. Delete the old launch configuration.

Fortunately, CloudFormation does this for you when you make changes to a launch configuration in a template. The following listing shows how to set up such a dynamic EC2 instance pool with the help of a CloudFormation template.

Listing 17.1 Auto-scaling group and launch configuration for a web app




```
# [...]
LaunchConfiguration:
  Type: 'AWS::AutoScaling::LaunchConfiguration'
  Properties:
    ImageId: 'ami-6057e21a'
    InstanceMonitoring: false
    InstanceType: 't2.micro'
    SecurityGroups:
    - webapp
    KeyName: mykey
    AssociatePublicIpAddress: true
    UserData:
      'Fn::Base64': !Sub |
        #!/bin/bash -x
        yum -y install httpd
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  Properties:
    TargetGroupARNs:
    - !Ref LoadBalancerTargetGroup
    LaunchConfigurationName: !Ref LaunchConfiguration
    MinSize: 2
    MaxSize: 4
    DesiredCapacity: 2
    HealthCheckGracePeriod: 300
    HealthCheckType: ELB
    VPCZoneIdentifier:
    - 'subnet-a55fafc'
    - 'subnet-fa224c5a'
# [...]
```

Auto-scaling groups are a useful tool if you need to start multiple virtual machines of the same kind across multiple availability zones. Additionally, an auto-scaling group replaces failed EC2 instances automatically.

17.2 Using metrics or schedules to trigger scaling

So far in this chapter, you've learned how to use an auto-scaling group and a launch configuration to manage virtual machines. With that in mind, you can change the desired capacity of the auto-scaling group manually, and new instances will be started or old instances will be terminated to reach the new desired capacity.

To provide a scalable infrastructure for a blogging platform, you need to increase and decrease the number of virtual machines in the pool automatically by adjusting the desired capacity of the auto-scaling group with scaling policies.

Many people surf the web during their lunch break, so you might need to add virtual machines every day between 11:00 a.m. and 1:00 p.m. You also need to adapt to unpredictable load patterns—for example, if articles hosted on your blogging platform are shared frequently through social networks.

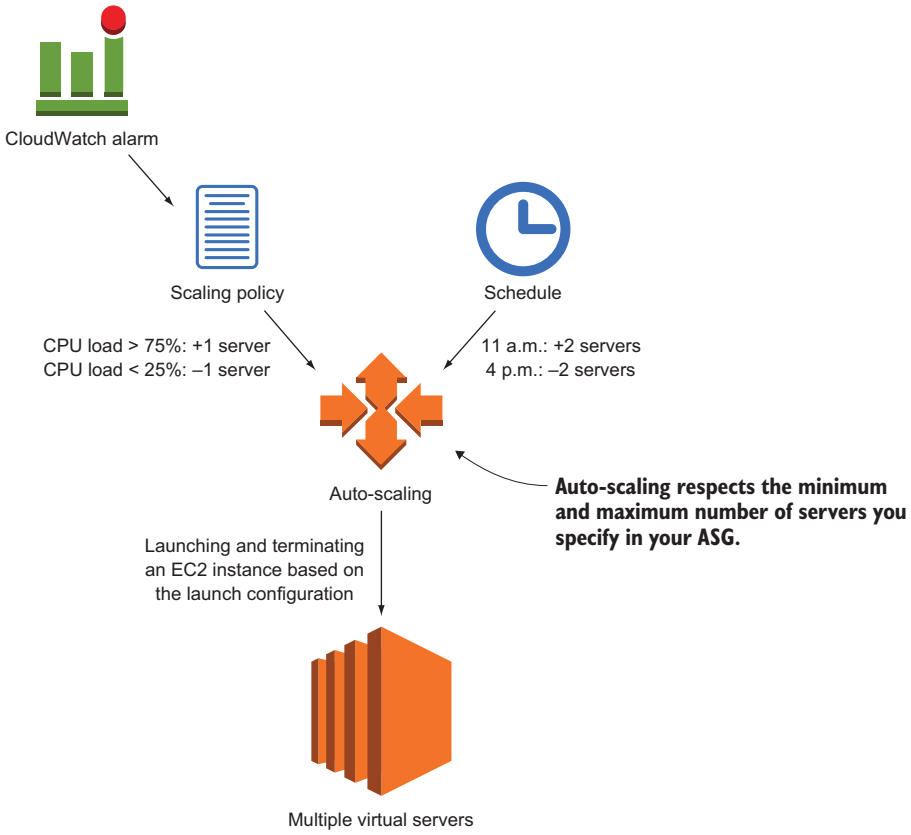


Figure 17.3 Triggering auto-scaling based on CloudWatch alarms or schedules

Figure 17.3 illustrates two different ways to change the number of virtual machines:

- *Defining a schedule.* The timing would increase or decrease the number of virtual machines according to recurring load patterns (such as decreasing the number of virtual machines at night).
- *Using a CloudWatch alarm.* The alarm will trigger a scaling policy to increase or decrease the number of virtual machines based on a metric (such as CPU usage or number of requests on the load balancer).

Scaling based on a schedule is less complex than scaling based on a CloudWatch metric, because it's difficult to find a metric to scale on reliably. On the other hand, scaling based on a schedule is less precise, as you have to over-provision your infrastructure to be able to handle unpredicted spikes in load.

17.2.1 Scaling based on a schedule

When operating a blogging platform, you might notice the following load patterns:

- *One-time actions*—Requests to your registration page increase heavily after you run a TV advertisement in the evening.
- *Recurring actions*—Many people seem to read articles during their lunch break, between 11:00 a.m. and 1:00 p.m.

Luckily, scheduled actions adjust your capacity with one-time or recurring actions. You can use different types of actions to react to both load pattern types.

The following listing shows a one-time scheduled action increasing the number of web servers at 12:00 UTC on Jan. 1, 2018. As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. The CloudFormation template for the WordPress example is located in `/chapter17/wordpress-schedule.yaml`.

Listing 17.2 Scheduling a one-time scaling action

```
OneTimeScheduledActionUp:
  Type: 'AWS::AutoScaling::ScheduledAction'      <-- Defining a scheduled action
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup   <-- Name of the auto-scaling group
    DesiredCapacity: 4                            <-- Set desired capacity to 4.
    StartTime: '2018-01-01T12:00:00Z'             <-- Change setting at 12:00 UTC on Jan. 1, 2018.
```

You can also schedule recurring scaling actions using cron syntax. The following example shows how to use two scheduled actions to increase the desired capacity during business hours (08:00 to 20:00 UTC) every day.

Listing 17.3 Scheduling a recurring scaling action that runs at 20:00 UTC every day

```
RecurringScheduledActionUp:
  Type: 'AWS::AutoScaling::ScheduledAction'    <-- Defining a scheduled action
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup
    DesiredCapacity: 4                         <-- Set desired capacity to 4.
    Recurrence: '0 8 * * *'                    <-- Increase capacity at 08:00 UTC every day.
 RecurringScheduledActionDown:
  Type: 'AWS::AutoScaling::ScheduledAction'
  Properties:
    AutoScalingGroupName: !Ref AutoScalingGroup
    DesiredCapacity: 2                         <-- Set desired capacity to 2.
    Recurrence: '0 20 * * *'                   <-- Decrease capacity at 20:00 UTC every day.
```

`Recurrence` is defined in Unix cron syntax format as shown here:

```
* * * * *
| | | | |
| | | | +- day of week (0 - 6) (0 Sunday)
| | | +--- month (1 - 12)
| | +----- day of month (1 - 31)
| +------- hour (0 - 23)
+--------- min (0 - 59)
```

We recommend using scheduled scaling actions whenever your infrastructure's capacity requirements are predictable—for example, an internal system used during work hours only, or a marketing action planned for a certain time.

17.2.2 Scaling based on CloudWatch metrics

Predicting the future is hard. Traffic will increase or decrease beyond known patterns from time to time. For example, if an article published on your blogging platform is heavily shared through social media, you need to be able to react to unplanned load changes and scale the number of EC2 instances.

You can adapt the number of EC2 instances to handle the current workload using CloudWatch alarms and scaling policies. CloudWatch helps monitor virtual machines and other services on AWS. Typically, services publish usage metrics to CloudWatch, helping you to evaluate the available capacity.

There are three types of scaling policies:

1. *Step scaling* allows more advanced scaling, as multiple scaling adjustments are supported, depending on how much the threshold you set has been exceeded.
2. *Target tracking* frees you from defining scaling steps and thresholds. You need only to define a target (such as CPU utilization of 70%) and the number of EC2 instances is adjusted accordingly.
3. *Simple scaling* is a legacy option which was replaced with *Step Scaling*.

All types of scaling policies use metrics and alarms to scale the number of EC2 instances based on the current workload. As shown in figure 17.4, the virtual machines publish metrics to CloudWatch constantly. A CloudWatch alarm monitors one of these metrics and triggers a scaling action if the defined threshold is reached. The scaling policy then increases or decreases the desired capacity of the auto-scaling group.

An EC2 instance publishes several metrics to CloudWatch by default: CPU, network, and disk utilization are the most important. Unfortunately, there is currently no metric for a virtual machine's memory usage. You can use these metrics to scale the number of VMs if a bottleneck is reached. For example, you can add EC2 instances if the CPU is working to capacity.

The following parameters describe a CloudWatch metric:

- `Namespace`—Defines the source of the metric (such as AWS/EC2)
- `Dimensions`—Defines the scope of the metric (such as all virtual machines belonging to an auto-scaling group)
- `MetricName`—Unique name of the metric (such as `CPUUtilization`)

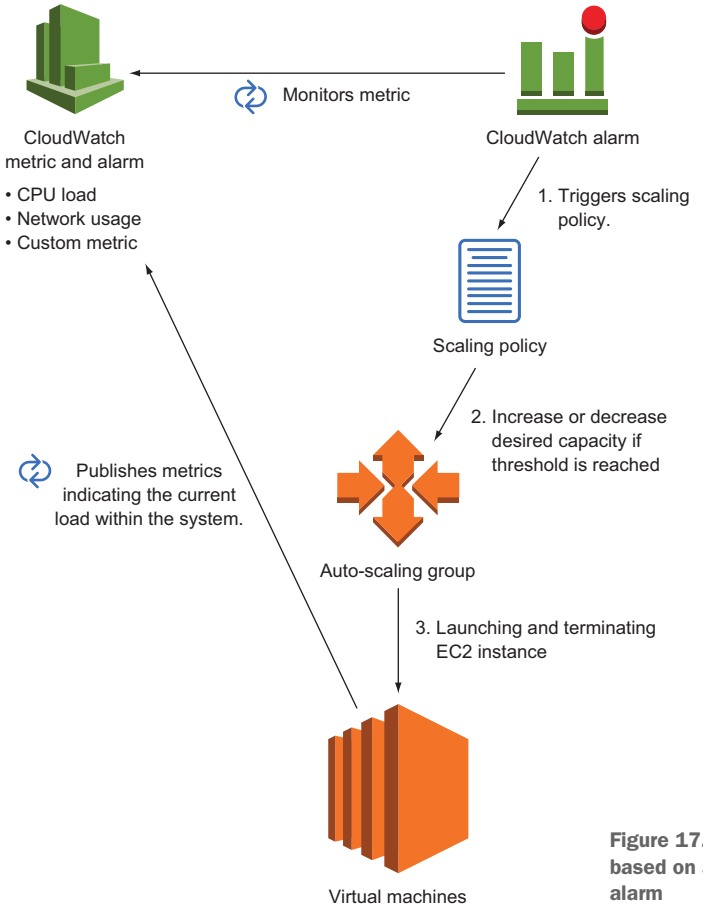


Figure 17.4 Triggering auto-scaling based on a CloudWatch metric and alarm

CloudWatch alarms are based on CloudWatch metrics. Table 17.3 explains the alarm parameters in detail.

Table 17.3 Parameters for a CloudWatch alarm that triggers scaling based on CPU utilization of all virtual machines belonging to an ASG

Context	Name	Description	Possible values
Condition	`Statistic`	Statistical function applied to a metric	`Average`, `Sum`, `Minimum`, `Maximum`, `SampleCount`
Condition	`Period`	Defines a time-based slice of values from a metric	Seconds (multiple of 60)
Condition	`EvaluationPeriods`	Number of periods to evaluate when checking for an alarm	Integer
Condition	`Threshold`	Threshold for an alarm	Number

Table 17.3 Parameters for a CloudWatch alarm that triggers scaling based on CPU utilization of all virtual machines belonging to an ASG *(continued)*

Context	Name	Description	Possible values
Condition	`ComparisonOperator`	Operator to compare the threshold against the result from a statistical function	`GreaterThanOrEqualToThreshold`, `GreaterThanThreshold`, `LessThanThreshold`, `LessThanOrEqualToThreshold`
Metric	`Namespace`	Source of the metric	`AWS/EC2` for metrics from the EC2 service
Metric	`Dimensions`	Scope of the metric	Depends on the metric; references the ASG for an aggregated metric over all associated EC2 instances
Metric	`MetricName`	Name of the metric	For example, `CPUUtilization`
Action	`AlarmActions`	Actions to trigger if the threshold is reached	Reference to the scaling policy

You can define alarms on many different metrics. You'll find an overview of all namespaces, dimensions, and metrics that AWS offers at http://mng.bz/8E0X. For example, you could scale based on the load balancer's metric counting the number of requests per target, or the networking throughput of your EC2 instances. You can also publish custom metrics—for example, metrics directly from your application like thread pool usage, processing times, or user sessions.

Scaling based on CPU load with VMs that offer burstable performance

Some virtual machines, such as instance family `t2`, offer burstable performance. These virtual machines offer a baseline CPU performance and can burst performance for a short time based on credits. If all credits are spent, the instance operates at the baseline. For a `t2.micro` instance, baseline performance is 10% of the performance of the underlying physical CPU.

Using virtual machines with burstable performance can help you react to load spikes. You save credits in times of low load, and spend credits to burst performance in times of high load. But scaling the number of virtual machines with burstable performance based on CPU load is tricky, because your scaling strategy must take into account whether your instances have enough credits to burst performance. Consider searching for another metric to scale (such as number of sessions) or using an instance type without burstable performance.

You've now learned how to use auto-scaling to adapt the number of virtual machines to the workload. It's time to bring this into action.

17.3 Decouple your dynamic EC2 instance pool

If you need to scale the number of virtual machines running your blogging platform based on demand, auto-scaling groups can help you provide the right number of uniform virtual machines, and scaling schedules or CloudWatch alarms can increase or decrease the desired number of EC2 instances automatically. But how can users reach the EC2 instances in the pool to browse the articles you're hosting? Where should the HTTP request be routed?

Chapter 15 introduced the concept of decoupling: synchronous decoupling with ELB, and asynchronous decoupling with SQS. If you want to use auto-scaling to grow and shrink the number of virtual machines, you need to decouple your EC2 instances from the clients, because the interface that's reachable from outside the system needs to stays the same no matter how many EC2 instances are working behind the scenes.

Figure 17.5 shows how to build a scalable system based on synchronous or asynchronous decoupling. A load balancer is acting as the entry point for synchronous decoupling, by distributing requests among a fleet of virtual machines. A message queue is used as the entry point for asynchronous requests. Messages from producers are stored in the queue. The virtual machines then poll the queue and process the messages asynchronously.

Decoupled and scalable applications require stateless servers. A stateless server stores any shared data remotely in a database or storage system. The following two examples implement the concept of a stateless server:

- *WordPress blog*—Decoupled with ELB, scaled with auto-scaling and CloudWatch based on CPU utilization, and data outsourced to a MySQL database (RDS) and a network filesystem (EFS).
- *URL2PNG taking screenshots of URLs*—Decoupled with a queue (SQS), scaled with auto-scaling and CloudWatch based on queue length, and data outsourced to a NoSQL database (DynamoDB) and an object store (S3).

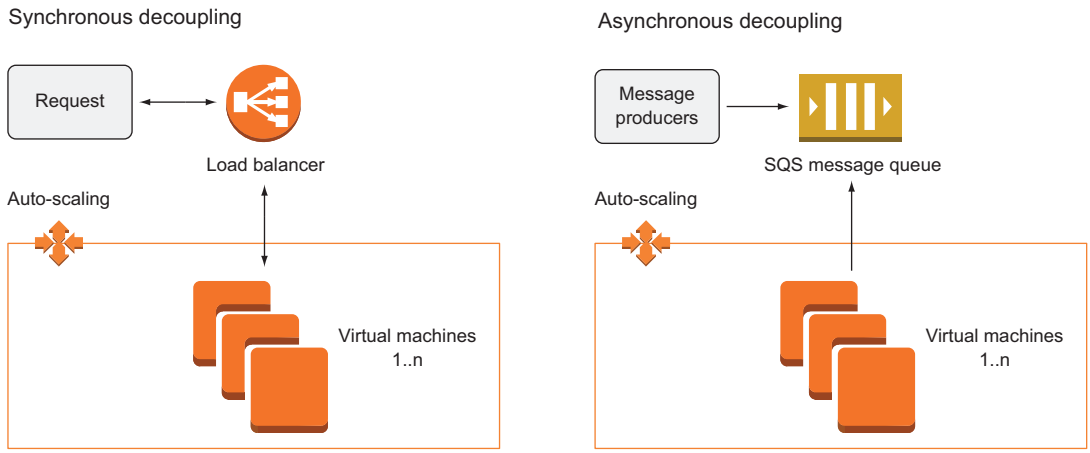


Figure 17.5 Decoupling allows you to scale the number of virtual machines dynamically.

17.3.1 Scaling a dynamic EC2 instance pool synchronously decoupled by a load balancer

Answering HTTP(S) requests is a synchronous task. If a user wants to use your web application, the web server has to answer the corresponding requests immediately. When using a dynamic EC2 instance pool to run a web application, it's common to use a load balancer to decouple the EC2 instances from user requests. The load balancer forwards HTTP(S) requests to multiple EC2 instances, acting as a single entry point to the dynamic EC2 instance pool.

Suppose your company has a corporate blog for publishing announcements and interacting with the community. You're responsible for hosting the blog. The marketing department complains about slow page speed and even timeouts in the evening, when traffic reaches its daily peak. You want to use the elasticity of AWS by scaling the number of EC2 instances based on the current workload.

Your company uses the popular blogging platform WordPress for its corporate blog. Chapters 2 and 11 introduced a WordPress setup based on EC2 instances and RDS (MySQL database). In this final chapter of the book, we'd like to complete the example by adding the ability to scale.

Figure 17.6 shows the final, extended WordPress example. The following services are used for this highly available scaling architecture:

- EC2 instances running Apache to serve WordPress, a PHP application
- RDS offering a MySQL database that's highly available through Multi-AZ deployment
- EFS storing PHP, HTML, and CSS files as well as user uploads such as images and videos
- ELB to synchronously decouple the web servers from visitors
- Auto-scaling and CloudWatch to scale the number of EC2 instances based on the current CPU load of all running virtual machines

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. The CloudFormation template for the WordPress example is located in `/chapter17/wordpress.yaml`.

Execute the following command to create a CloudFormation stack that spins up the scalable WordPress setup. Replace *$Password* with your own password consisting of 8 to 30 letters and digits.

```
$ aws cloudformation create-stack --stack-name wordpress \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter17/wordpress.yaml --parameters \
➥ ParameterKey=WordpressAdminPassword,ParameterValue=$Password \
➥ --capabilities CAPABILITY_IAM
```

It will take up to 15 minutes for the stack to be created. This is a perfect time to grab some coffee or tea. Log in to the AWS Management Console, and navigate to the AWS CloudFormation service to monitor the process of the CloudFormation stack named

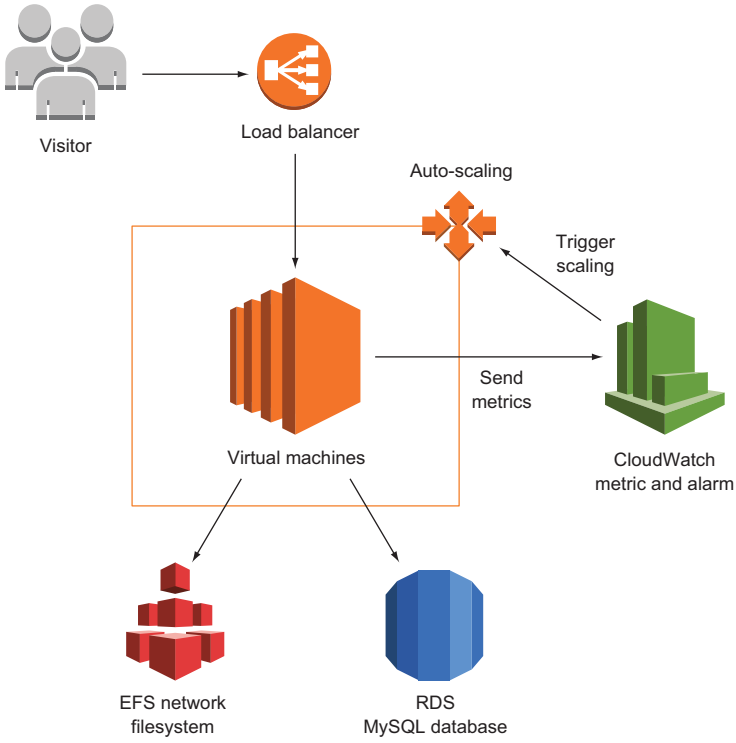


Figure 17.6 Auto-scaling web servers running WordPress, storing data on RDS and EFS, decoupled with a load balancer scaling based on load

`wordpress`. You have time to look through the most important parts of the CloudFormation template, shown in the following two listings.

Listing 17.4 Creating a scalable, HA WordPress setup (part 1)

```
LaunchConfiguration:
  Type: 'AWS::AutoScaling::LaunchConfiguration'
  Metadata: # [...]
  Properties:
    AssociatePublicIpAddress: true
    ImageId: 'ami-6057e21a'
    InstanceMonitoring: false
    InstanceType: 't2.micro'
    SecurityGroups:
    - !Ref WebServerSecurityGroup
    KeyName: !Ref KeyName
    UserData: # [...]
AutoScalingGroup:
  Type: 'AWS::AutoScaling::AutoScalingGroup'
  DependsOn:
```

Creates a launch configuration for auto-scaling

Image (AMI) from which to start a virtual machine

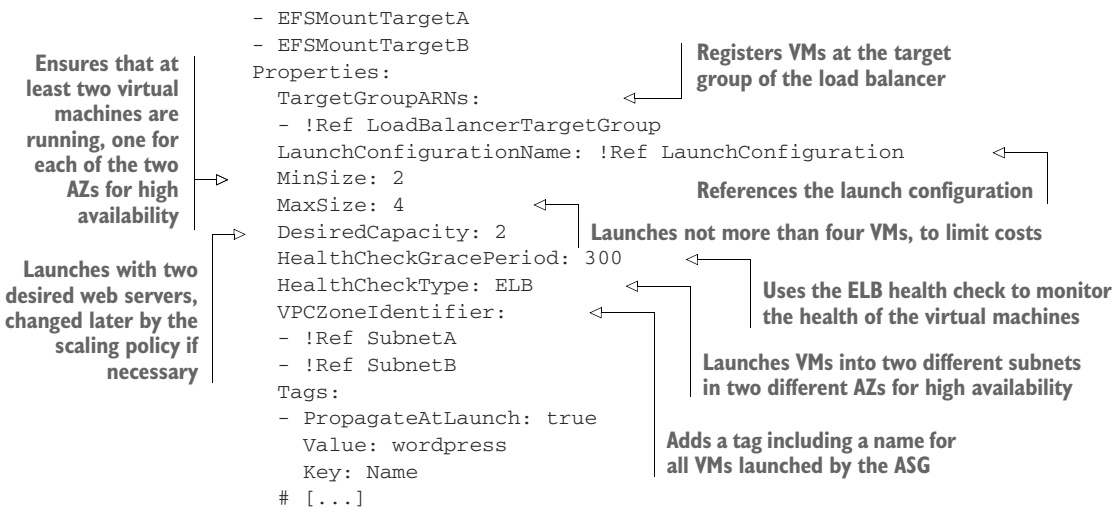
Size of the virtual machine

Key pair for SSH access

Security group with firewall rules for virtual machines

Script to install and configure WordPress automatically

Creates an auto-scaling group



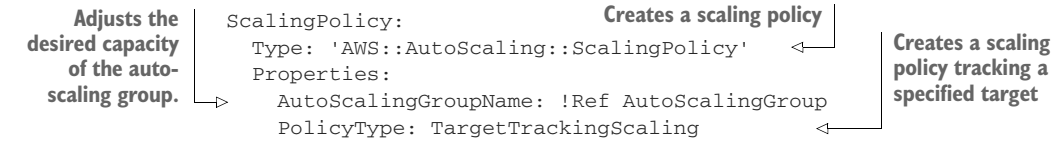
You will learn how to create CloudWatch alarms for scaling in the next example. For now, we are using a target tracking scaling policy that creates CloudWatch alarms automatically in the background. A target tracking scaling policy works like the thermostat in your home: you define the target and the thermostat constantly adjusts the heating power to reach the target.

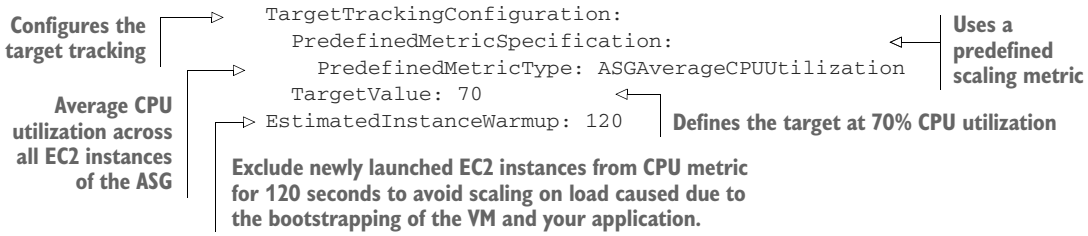
Predefined metric specifications for the use with target tracking are:

- `ASGAverageCPUUtilization` to scale based on the average CPU utilization among all instances within an auto-scaling group.
- `ALBRequestCountPerTarget` to scale based on the number of requests forwarded from the Application Load Balancer (ALB) to a target.
- `ASGAverageNetworkIn` and `ASGAverageNetworkOut` to scale based on the average number of bytes received or sent.

In some cases, scaling based on CPU utilization, request count per target, or network throughput does not work. For example, you might have another bottleneck you need to scale on, such as disk I/O. Any CloudWatch metric can be used for target tracking as well. There is only one requirement: adding or removing instances must affect the metric proportionally. For example, request latency is not a valid metric for target tracking, as adjusting the number of instances does not affect the request latency directly.

Listing 17.5 Creating a scalable, HA WordPress setup (part 2)





Follow these steps after the CloudFormation stack reaches the state `CREATE_COMPLETE` to create a new blog post containing an image:

1. Select the CloudFormation stack `wordpress`, and switch to the Outputs tab.
2. Open the link shown for key `URL` with a modern web browser.
3. Search for the Log In link in the navigation bar, and click it.
4. Log in with username `admin` and the password you specified when creating the stack with the CLI.
5. Click Posts in the menu on the left.
6. Click Add New.
7. Type in a title and text, and upload an image to your post.
8. Click Publish.
9. Go back to the blog by clicking on the View Post link.

Now you're ready to scale. We've prepared a load test that will send 500,000 requests to the WordPress setup within a few minutes. Don't worry about costs: the usage is covered by the Free Tier. After three minutes, new virtual machines will be launched to handle the load. The load test takes 10 minutes. Another 15 minutes later, the additional VMs will disappear. Watching this is fun; you shouldn't miss it.

NOTE If you plan to do a big load test, consider the AWS Acceptable Use Policy at https://aws.amazon.com/aup and ask for permission before you begin (see also https://aws.amazon.com/security/penetration-testing).

Simple HTTP load test

We're using a tool called *Apache Bench* to perform a load test of the WordPress setup. The tool is part of the `httpd-tools` package available from the Amazon Linux package repositories.

Apache Bench is a basic benchmarking tool. You can send a specified number of HTTP requests by using a specified number of threads. We're using the following command for the load test, to send 500,000 requests to the load balancer using 15 threads. The load test is limited to 600 seconds and we're using a connection timeout of 120 seconds. Replace *$UrlLoadBalancer* with the URL of the load balancer:

```
$ ab -n 500000 -c 15 -t 300 -s 120 -r $UrlLoadBalancer
```

Update the CloudFormation stack with the following command to start the load test:

```
$ aws cloudformation update-stack --stack-name wordpress \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter17/wordpress-loadtest.yaml \
➥ --parameters ParameterKey=WordpressAdminPassword,UsePreviousValue=true  \
➥ --capabilities CAPABILITY_IAM
```

Watch for the following things to happen, using the AWS Management Console:

1. Open the CloudWatch service, and click Alarms on the left.
2. When the load test starts, the alarm called `TargetTracking-wordpress-AutoScalingGroup-*-AlarmHigh-*` will reach the `ALARM` state after about 3 minutes.
3. Open the EC2 service, and list all EC2 instances. Watch for two additional instances to launch. At the end, you'll see five instances total (four web servers and the EC2 instance running the load test).
4. Go back to the CloudWatch service, and wait until the alarm named `TargetTracking-wordpress-AutoScalingGroup-*-AlarmLow-*` reaches the `ALARM` state.
5. Open the EC2 service, and list all EC2 instances. Watch for the two additional instances to disappear. At the end, you'll see three instances total (two web servers and the EC2 instance running the load test).

The entire process will take about 20 minutes.

You've watched auto-scaling in action: your WordPress setup can now adapt to the current workload. The problem with pages loading slowly or even timeouts in the evening is solved.

Cleaning up

Execute the following commands to delete all resources corresponding to the WordPress setup:

```
$ aws cloudformation delete-stack --stack-name wordpress
```

17.3.2 Scaling a dynamic EC2 instances pool asynchronously decoupled by a queue

Imagine that you're developing a social bookmark service where users can save and share their links. Offering a preview that shows the website being linked to is an important feature. But the conversion from URL to PNG is causing high load during the evening, when most users add new bookmarks to your service. Because of that, customers are dissatisfied with your application's slow response times.

You will learn how to dynamically scale a fleet of EC2 instances to asynchronously generate screenshots of URLs in the following example. Doing so allows you to

guarantee low response times at any time because the load-intensive workload is isolated into background jobs.

Decoupling a dynamic EC2 instance pool asynchronously offers an advantage if you want to scale based on workload: because requests don't need to be answered immediately, you can put requests into a queue and scale the number of EC2 instances based on the length of the queue. This gives you an accurate metric to scale, and no requests will be lost during a load peak because they're stored in a queue.

To handle the peak load in the evening, you want to use auto-scaling. To do so, you need to decouple the creation of a new bookmark and the process of generating a preview of the website. Chapter 12 introduced an application called URL2PNG that transforms a URL into a PNG image. Figure 17.7 shows the architecture, which consists of an SQS queue for asynchronous decoupling as well as S3 for storing generated images. Creating a bookmark will trigger the following process:

1. A message is sent to an SQS queue containing the URL and the unique ID of the new bookmark.
2. EC2 instances running a Node.js application poll the SQS queue.
3. The Node.js application loads the URL and creates a screenshot.
4. The screenshot is uploaded to an S3 bucket, and the object key is set to the unique ID.
5. Users can download the screenshot directly from S3 using the unique ID.

A CloudWatch alarm is used to monitor the length of the SQS queue. If the length of the queue reaches five, an additional virtual machine is started to handle the workload. When the queue length goes below five, another CloudWatch alarm decreases the desired capacity of the auto-scaling group.

The code is in the book's code repository on GitHub at https://github.com/AWSinAction/code2. The CloudFormation template for the URL2PNG example is located at chapter17/url2png.yaml.

Execute the following command to create a CloudFormation stack that spins up the URL2PNG application. Replace *$ApplicationID* with a unique ID for your application (such as url2png-andreas):

```
$ aws cloudformation create-stack --stack-name url2png \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter17/url2png.yaml \
➥ --parameters ParameterKey=ApplicationID,ParameterValue=$ApplicationID \
➥ --capabilities CAPABILITY_IAM
```

It will take up to five minutes for the stack to be created. Log in to the AWS Management Console, and navigate to the AWS CloudFormation service to monitor the process of the CloudFormation stack named `url2png`.

We're using the length of the SQS queue to scale the number of EC2 instances. As the number of messages in the queue does not correlate with the number of EC2

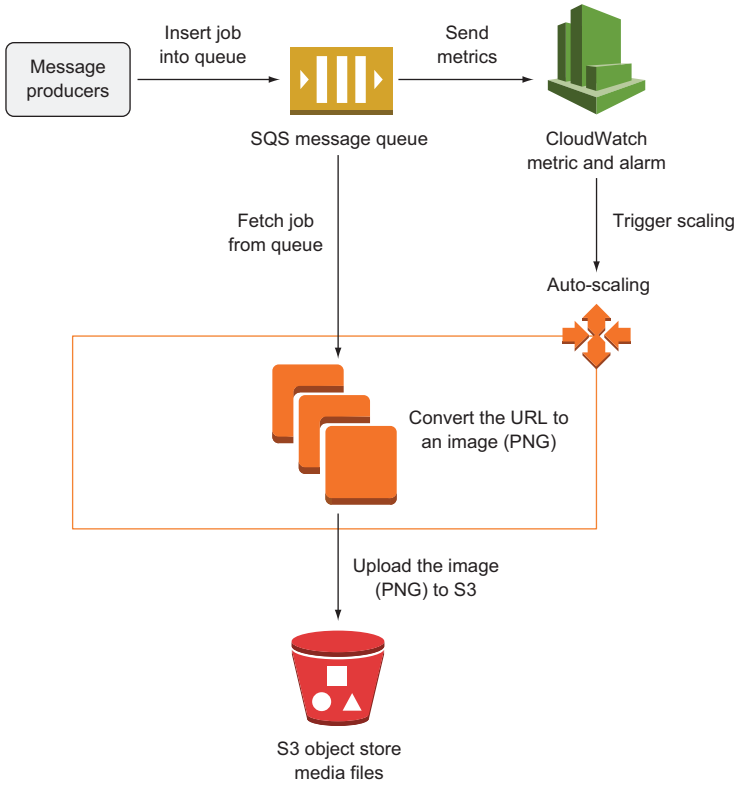


Figure 17.7 Auto-scaling virtual machines that convert URLs into images, decoupled by an SQS queue

instances processing messages from the queue, it is not possible to use a target tracking policy. Therefore, you will use a step scaling policy in this scenario.

Listing 17.6 Monitoring the length of the SQS queue

```
# [...]
HighQueueAlarm:
  Type: 'AWS::CloudWatch::Alarm'
  Properties:
    EvaluationPeriods: 1
    Statistic: Sum
    Threshold: 5
    AlarmDescription: 'Alarm if queue length is higher than 5.'
    Period: 300
    AlarmActions:
    - !Ref ScalingUpPolicy
    Namespace: 'AWS/SQS'
    Dimensions:
    - Name: QueueName
      Value: !Sub '${SQSQueue.QueueName}'
```

Number of time periods to evaluate when checking for an alarm

Sums up all values in a period

Alarms if the threshold of 5 is reached

Uses a period of 300 seconds because SQS metrics are published every 5 minutes

Increases the number of desired instances by 1 through the scaling policy

The metric is published by the SQS service.

The queue, referenced by name, is used as the dimension of the metric.

Alarms if the sum of the values within the period is greater than the threshold of 5

```
    ComparisonOperator: GreaterThanThreshold
    MetricName: ApproximateNumberOfMessagesVisible
# [...]
```

The metric contains an approximate number of messages pending in the queue.

The CloudWatch alarm triggers a scaling policy. The scaling policy defines how to scale. To keep things simple, we are using a step scaling policy with only a single step. Add additional steps if you want to react to a threshold breach in a more fine-grained way.

Listing 17.7 A step scaling policy which adds one more instance to an ASG

Creates a scaling policy

Attaches the scaling policy to the auto-scaling group

Creates a scaling policy of type step scaling

Defines the scaling steps. We use a single step in this example.

```
# [...]
ScalingUpPolicy:
  Type: 'AWS::AutoScaling::ScalingPolicy'
  Properties:
    AdjustmentType: 'ChangeInCapacity'
    AutoScalingGroupName: !Ref AutoScalingGroup
    PolicyType: 'StepScaling'
    MetricAggregationType: 'Average'
    EstimatedInstanceWarmup: 60
    StepAdjustments:
    - MetricIntervalLowerBound: 0
      ScalingAdjustment: 1
# [...]
```

The scaling policy increases the capacity by an absolute number.

The aggregation type used when evaluating the steps, based on the metric defined within the CloudWatch alarm that triggers the scaling policy

The metrics of a newly launched instance are ignored for 60 seconds while it boots up.

The scaling step is valid from the alarms threshold to infinity.

Increase the desired capacity of the ASG by 1.

To scale down the number of instances when the queue is empty, a CloudWatch alarm and scaling policy with the opposite values needs to be defined.

You're ready to scale. We've prepared a load test that will quickly generate 250 messages for the URL2PNG application. A virtual machine will be launched to process jobs from the SQS queue. After a few minutes, when the load test is finished, the additional virtual machine will disappear.

Update the CloudFormation stack with the following command to start the load test:

```
$ aws cloudformation update-stack --stack-name url2png \
➥ --template-url https://s3.amazonaws.com/\
➥ awsinaction-code2/chapter17/url2png-loadtest.yaml \
➥ --parameters ParameterKey=ApplicationID,UsePreviousValue=true \
➥ --capabilities CAPABILITY_IAM
```

Watch for the following things to happen, with the help of the AWS Management Console:

1. Open the CloudWatch service, and click Alarms at left.
2. When the load test starts, the alarm called `url2png-HighQueueAlarm-*` will reach the `ALARM` state after a few minutes.
3. Open the EC2 service, and list all EC2 instances. Watch for an additional instance to launch. At the end, you'll see three instances total (two workers and the EC2 instance running the load test).
4. Go back to the CloudWatch service, and wait until the alarm named `url2png-LowQueueAlarm-*` reaches the `ALARM` state.
5. Open the EC2 service, and list all EC2 instances. Watch for the additional instance to disappear. At the end, you'll see two instances total (one worker and the EC2 instance running the load test).

The entire process will take about 15 minutes.

You've watched auto-scaling in action. The URL2PNG application can now adapt to the current workload, and the problem with slowly generated screenshots has been solved.

Cleaning up

Execute the following commands to delete all resources corresponding to the URL2PNG setup, remembering to replace *$ApplicationID*:

```
$ aws s3 rm s3://$ApplicationID --recursive
$ aws cloudformation delete-stack --stack-name url2png
```

Whenever distributing an application among multiple EC2 instances, you should use an auto-scaling group. Doing so allows you to spin up identical instances with ease. You get the most out of the possibilities of the cloud when scaling the number of instances based on a schedule or a metric depending on the load pattern.

Summary

- You can use auto-scaling to launch multiple identical virtual machines by using a launch configuration and an auto-scaling group.
- EC2, SQS, and other services publish metrics to CloudWatch (CPU utilization, queue length, and so on).
- CloudWatch alarms can change the desired capacity of an auto-scaling group. This allows you to increase the number of virtual machines based on CPU utilization or other metrics.
- Virtual machines need to be stateless if you want to scale them according to your current workload.

- To distribute load among multiple virtual machines, synchronous decoupling with the help of a load balancer or asynchronous decoupling with a message queue is necessary.

That's it! You have mastered the end game: scaling your infrastructure dynamically. Kudos, you have learned about and experienced the most important aspects of Amazon Web Services. We wish you all the best for moving your first production workload to the cloud.

index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

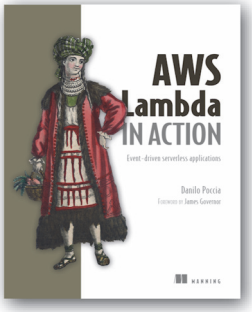
Y

RELATED MANNING TITLES

Learn Amazon Web Services in a Month of Lunches

by David Clinton

ISBN: 9781617294440
328 pages, $39.99
August 2017



AWS Lambda in Action

Event-driven serverless applications

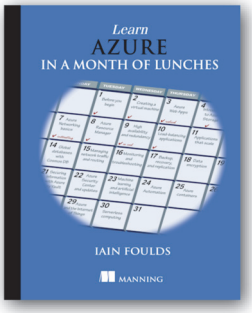
by Danilo Poccia

ISBN: 9781617293719
384 pages, $49.99
November 2016

Google Cloud Platform in Action

by JJ Geewax

ISBN: 9781617293528
632 pages, $59.99
September 2018



Learn Azure in a Month of Lunches

by Iain Foulds

ISBN: 9781617295171
375 pages, $44.99
September 2018

For ordering information go to www.manning.com

Storage & Database

Abbr.	Name	Description	Where
S3	Amazon Simple Storage Service	Object store to save data without any size restrictions	8
	Amazon Glacier	Inexpensive data archive solution	8.4
EBS	Amazon Elastic Block Store	Network attached block level storage for EC2 instances	9.1
	Amazon EC2 Instance Store	Block level storage for EC2 instances	9.2
EFS	Amazon Elastic File System	Scalable network file system based on NFSv4	10
RDS	Amazon Relational Database Service	MySQL, Oracle Database, Microsoft SQL Server, or PostgreSQL	11
	Amazon DynamoDB	Proprietary NoSQL key-value database with limited but powerful options for queries	13
	Amazon ElastiCache	In-memory data store based on Redis or Memcached typically used to cache data	12

Architecting on AWS

Abbr.	Name	Description	Where
AZ	Availability Zone	Group of isolated data centers within a region	14.2
ASG	Amazon EC2 auto-scaling group	Observes a fleet of servers: Replaces faulty servers and increases/decreases the size of the fleet based on external triggers like CPU usage	17
	Amazon CloudWatch	Keeps track of metrics and triggers alarms when a threshold is reached	14.1
ELB	Elastic Load Balancing	Load balancer for EC2 instances	15.1
ALB	Application Load Balancer	Layer 7 load balancer with HTTP and HTTPS support	15.1
SQS	Amazon Simple Queue Service	Message queue	15.2

CLOUD/AWS

Amazon Web Services IN ACTION Second Edition

Michael Wittig • Andreas Wittig

The largest and most mature of the cloud platforms, AWS offers over 100 prebuilt services, practically limitless compute resources, bottomless secure storage, as well as top-notch automation capabilities. This book shows you how to develop, host, and manage applications on AWS.

Amazon Web Services in Action, Second Edition is a comprehensive introduction to deploying web applications in the AWS cloud. You'll find clear, relevant coverage of all essential AWS services, with a focus on automation, security, high availability, and scalability. This thoroughly revised edition covers the latest additions to AWS, including serverless infrastructure with AWS Lambda, sharing data with EFS, and in-memory storage with ElastiCache.

What's Inside

- Completely revised bestseller!
- Secure and scale distributed applications
- Deploy applications on AWS
- Design for failure to achieve high availability
- Automate your infrastructure

Written for mid-level developers and DevOps engineers.

Andreas and **Michael Wittig** are software engineers and DevOps consultants focused on AWS. Together, they migrated the first bank in Germany to AWS in 2013.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/amazon-web-services-in-action-second-edition

"Slices through the complexity of AWS using examples and visuals to cement knowledge in the minds of readers."

—From the Foreword by Ben Whaley
AWS community hero and author

"The authors' ability to explain complex concepts is the real strength of the book."

—Antonio Pessolano
Consoft Sistemi

"Useful examples, figures, and sources to help you learn efficiently."

—Christof Marte, Daimler-Benz

"Does a great job of explaining some of the key services in plain English so you have the knowledge necessary to dig deeper."

—Ryan Burrows
Rooster Park Consulting

ISBN-13: 978-1-61729-511-9
ISBN-10: 1-61729-511-6
55499
9 781617 295119

MANNING $54.99 / Can $72.99 [INCLUDING eBOOK]