Survive Your Success

Programming

Amazon EC2

O'REILLY®

Jurg van Vliet
& Flavia Paganelli

www.it-ebooks.info

Programming Amazon EC2

Jurg van Vliet and Flavia Paganelli

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

www.it-ebooks.info

Programming Amazon EC2
by Jurg van Vliet and Flavia Paganelli

Copyright © 2011 I-MO BV. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Julie Steele
Production Editor: Adam Zaremba
Copyeditor: Amy Thomson
Proofreader: Emily Quill
Indexer: John Bickelhaupt
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
February 2011: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Programming Amazon EC2*, the image of a bushmaster snake, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-39368-7

[LSI]

1297365147

www.it-ebooks.info

Table of Contents

Foreword . . . . **ix**

Preface . . . . **xiii**

1. Introducing AWS **1**

From 0 to AWS 1
- Biggest Problem First 2
- Infinite Storage 3
- Computing Per Hour 4
- Very Scalable Data Store 5
- Optimizing Even More 6
- Going Global 7

Growing into Your Application 7
- Start with Realistic Expectations 7
- Simply Small 8
- Growing Up 9
- Moving Out 10

"You Build It, You Run It" 11
- Individuals and Interactions: One Team 11
- Working Software: Shared Responsibility 12
- Customer Collaboration: Evolve Your Infrastructure 13
- Responding to Change: Saying Yes with a Smile 13

In Short 14

2. Starting with EC2, RDS, and S3/CloudFront **15**

Setting Up Your Environment 16
- Your AWS Account 16
- Command-Line Tools 17
- AWS Management Console 19
- Other Tools 20

Choosing Your Geographic Location, Regions, and Availability Zones 21

www.it-ebooks.info

Choosing an Architecture 21
Creating the Rails Server on EC2 22
Creating a Key Pair 23
Finding a Suitable AMI 23
Setting Up the Web/Application Server 24
RDS Database 35
Creating an RDS Instance (Launching the DB Instance Wizard) 36
Is This All? 39
S3/CloudFront 41
Setting Up S3 and CloudFront 41
Static Content to S3/CloudFront 43
Making Backups of Volumes 45
Installing the Tools 46
Running the Script 46
In Short 49

3. Growing with S3, ELB, Auto Scaling, and RDS ... 51
Preparing to Scale 52
Setting Up the Tools 54
S3 for File Uploads 54
User Uploads for Kulitzer (Rails) 54
Elastic Load Balancing 55
Creating an ELB 56
Difficulties with ELB 59
Auto Scaling 60
Setting Up Auto Scaling 60
Auto Scaling in Production 64
Scaling a Relational Database 66
Scaling Up (or Down) 66
Scaling Out 68
Tips and Tricks 69
Elastic Beanstalk 70
In Short 72

4. Decoupling with SQS, SimpleDB, and SNS ... 73
SQS 73
Example 1: Offloading Image Processing for Kulitzer (Ruby) 74
Example 2: Priority PDF Processing for Marvia (PHP) 77
Example 3: Monitoring Queues in Decaf (Java) 81
SimpleDB 85
Use Cases for SimpleDB 87
Example 1: Storing Users for Kulitzer (Ruby) 88
Example 2: Sharing Marvia Accounts and Templates (PHP) 91

www.it-ebooks.info

Example 3: SimpleDB in Decaf (Java) 95
SNS 99
Example 1: Implementing Contest Rules for Kulitzer (Ruby) 100
Example 2: PDF Processing Status (Monitoring) for Marvia (PHP) 105
Example 3: SNS in Decaf (Java) 108
In Short 111

5. Managing the Inevitable Downtime 113
Measure 114
Up/Down Alerts 114
Monitoring on the Inside 114
Monitoring on the Outside 118
Understand 122
Why Did I Lose My Instance? 122
Spikes Are Interesting 122
Predicting Bottlenecks 124
Improvement Strategies 124
Benchmarking and Tuning 124
The Merits of Virtual Hardware 125
In Short 126

6. Improving Your Uptime 129
Measure 129
EC2 130
ELB 131
RDS 132
Using Dimensions from the Command Line 133
Alerts 134
Understand 136
Setting Expectations 136
Viewing Components 137
Improvement Strategies 138
Planning Nonautoscaling Components 138
Tuning Auto Scaling 138
In Short 138

7. Managing Your Decoupled System 141
Measure 141
S3 142
SQS 142
SimpleDB 149
SNS 152
Understand 153

www.it-ebooks.info

Imbalances 154
Bursts 154
Improvement Strategies 154
Queues Neutralize Bursts 155
Notifications Accelerate 155
In Short 156

8. And Now... 157
Other Approaches 157
Private/Hybrid Clouds 158
Thank You 158

Index 159

www.it-ebooks.info

Foreword

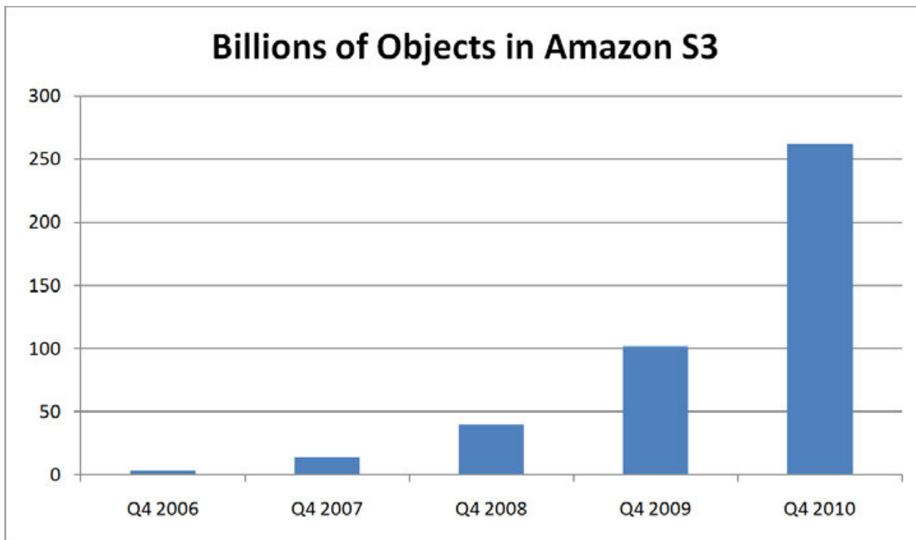
March 14, 2006, was an important day, even though it is unlikely that it will ever become more than a footnote in some history books. On that day, Amazon Web Services launched the first of its utility computing services: the Amazon Simple Storage Service (Amazon S3). In my eyes that was the day that changed the way IT was done; it gave everyone access to an ultra-reliable and highly scalable storage service without having to invest tens of thousands of dollars for an exclusive enterprise storage solution. And even better, the service sat directly on the Internet, and objects were directly HTTP addressable.

The motivation behind the launch of the service was simple: the AWS team had asked itself what innovation could happen if it could give everyone access to the same scalable and reliable technologies that were available to Amazon engineers. A student in her dorm room could have an idea that could become the next Amazon or the next Google, and the only thing that would hold her back was access to the resources needed to fulfill that potential. AWS aimed at removing these barriers and constraints so people could unleash their innovation and focus on building great new products instead of having to invest in infrastructure both intellectually and financially.

Today, Amazon S3 has grown to store more than 260 billion objects and routinely runs more than 200,000 storage operations per second. The service has become a fundamental building block for many applications, from enterprise ERP log files to blog storage, streaming videos, software distribution, medical records, and astronomy data.

By routinely running over 200,000 storage operations per second, Amazon S3 is a marvel of technology under the covers. It is designed to support a wide range of usage scenarios and is optimized in very innovative ways to make sure every customer gets great service, regardless of whether he is streaming videos or just housing some home photos. One of my colleagues had a great analogy about how the Amazon S3 software had to evolve: it was like starting with a single-engine Cessna that had to be rebuilt into a Boeing 747 while continuing to fly and continuously refueling, and with passengers that changed planes without noticing it. The Amazon S3 team has done a great job of making the service something millions and millions of people rely on every day.

www.it-ebooks.info

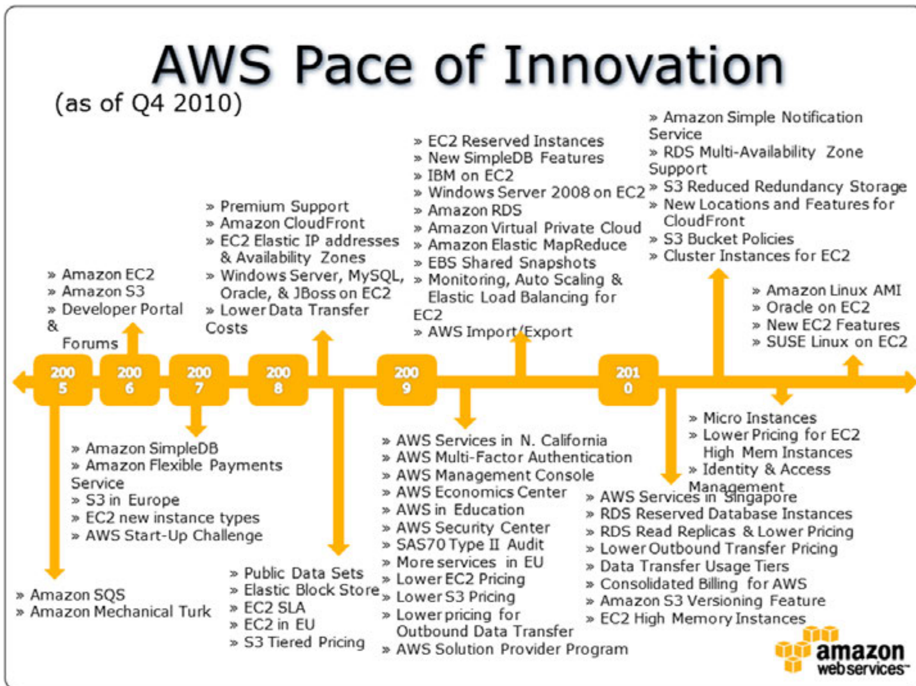


Following Amazon S3, we launched Amazon Simple Queue Service (Amazon SQS), and then Amazon Elastic Compute Cloud (Amazon EC2) just a few months later. These services demonstrated the power of what we have come to call Cloud Computing: access to highly reliable and scalable infrastructure with a utility payment model that drives innovation and dramatically shortens time to market for new products. Many CIOs have told me that while their first motivation to start using AWS was driven by the attractive financial model, the main reason for staying with AWS is that it has made their IT departments agile and allowed them to become enablers of innovation within their organization.

The AWS platform of technology infrastructure services and features has grown rapidly since that day in March 2006, and we continue to keep that same quick pace of innovation and relentless customer focus today.

Although AWS, as well as its ecosystem, has launched many tools that make using the services really simple, at its core it is still a fully programmable service with incredible power, served through an API. Jurg and Flavia have done a great job in this book of building a practical guide for how to build real systems using AWS. Their writing is based on real experiences using each and every one of the AWS services, and their advice is rooted in building foundations upon which applications on the AWS platform can scale and remain reliable. I first came in contact with them when they were building Decaf, an Android application used to control your AWS resources from your mobile device. Since then, I have seen them help countless customers move onto the AWS platform, and also help existing customers scale better and become more reliable while taking advantage of the AWS elasticity to drive costs down. Their strong customer focus makes them great AWS partners.

www.it-ebooks.info



The list of services and features from these past years may seem overwhelming, but our customers continue to ask for more ways to help us remove nonessential infrastructure tasks from their plate so that they can focus on what really matters to them: delivering better products and services to their customers.

AWS will continue to innovate on behalf of our customers, and there are still very exciting things to come.

—Werner Vogels
VP & CTO at Amazon.com

www.it-ebooks.info

Preface

Thank you for picking up a copy of this book. Amazon Web Services (AWS) has amazed everyone: Amazon has made lots of friends, and all its "enemies" are too busy admiring AWS to do much fighting back. At the moment, there is no comparable public Infrastructure as a Service (IaaS); AWS offers the services at a scale that has not been seen before. We wrote this book so you can get the most out of AWS' services. If you come from conventional hardware infrastructures, once you are on AWS, you won't want to go back.

AWS is not easy; it combines skills of several different (established) crafts. It is different from traditional systems administration, and it's not just developing a piece of software. If you have practiced one or both of these skills, all you need is to be inquisitive and open to learning.

Our background is in software engineering. We are computer scientists with extensive software engineering experience in all sorts of different fields and organizations. But the cloud in general and AWS in particular caught our interest some years ago. We got serious about this by building Decaf, an Android smartphone application that manages Amazon EC2 (Elastic Compute Cloud) accounts. We were finalists in the Android Developer Challenge in 2009. We will use Decaf to illustrate various AWS services and techniques throughout this book.

Around the same time, in early 2010, we decided we wanted to build applications on AWS. We founded 9Apps and set out to find a select group of partners who shared our development interests. Our expertise is AWS, and our responsibility is to keep it running at all times. We design, build, and operate these infrastructures.

Much of our experience comes from working with these teams and building these applications, and we will use several of them as examples throughout the book. Here is a short introduction to the companies whose applications we will use:

Directness

Directness helps customers connect brands to businesses. With a set of tools for making surveys and collecting, interpreting, and presenting consumers' feedback, this application is very successful in its approach and works with a number of international firms. The problem is scaling the collection of customer responses,

www.it-ebooks.info

transforming it into usable information, and presenting it to the client. Directness can only grow if we solve this problem.

Kulitzer
: Kulitzer is a web application that allows users to organize creative contests. Users can invite participants to enter the contest, an audience to watch, and a jury to pick a winner. Technically, you can consider Kulitzer a classical consumer web app.

Layar
: Layar is an augmented reality (AR) smartphone browser that is amazing everyone. This application enriches the user's view of the world by overlapping different objects or information in the camera view, relevant to the location. For example, users can see what people have been tweeting near them, the houses that are for sale in the neighborhood, or tourist attractions near where they are walking.

  The Layar application continues to win prize after prize, and is featured in many technical and mainstream publications. Layar started using Google App Engine for its servers, but for several reasons has since moved to AWS.

Marvia
: Ever needed to create some "print ready" PDFs? It's not an easy task. You probably needed desktop publishing professionals and the help of a marketing agency, all for a significant price tag. Marvia is an application that can dramatically reduce the effort and cost involved in PDF creation. It allows you to create reusable templates with a drag-and-drop web application. Or you can integrate your own system with Marvia's API to automate the generation of leaflets and other material.

Publitas
: Publitas does the opposite of what Marvia does, in a way. It lets you transform your traditional publication material to an online experience. The tool, called ePublisher, is very feature-rich and is attracting a lot of attention. You can input your material in PDF format to the application and it will generate online content. You can then enrich the content with extra functionality, such as supporting sharing in social networks and adding music, video, search, and print. The challenge with the Publitas software is that its existing customers are established and well-known businesses that are sometimes already so powerful that exposure ratings resemble those of a mass medium like television.

Audience

Of course, we welcome all readers of this book, and we hope it inspires you to get into AWS and utilize it in the best possible way to be successful. But we set out to write this book with a particular purpose: to be an AWS guide for building and growing applications from small to "Internet scale." It will be useful if you want to host your blog or small web application, but it will also help you grow like Zynga did with Farmville. (Some say Zynga is the fastest growing company in the world.)

www.it-ebooks.info

This book does not focus on detail; for example, we are not going to tell you exactly which parameters each command receives, and we are not going to list all the available commands. But we will show you the approach and implementation. We rely on examples to illustrate the concepts and to provide a starting point for your own projects. We try to give you a sense of all AWS functionality, which would be nearly impossible if we were to show the details of every feature.

To get the most out of this book, you should be comfortable with the command line, and having experience writing software will be useful for some of the chapters. And it certainly wouldn't hurt if you know what Ubuntu is (or CentOS or Windows 2003, for that matter) and how to install software. But most of all, you should simply be curious about what you can do with AWS. There's often more than one way of doing things, and since AWS is so new, many of those ways have not yet been fully explored.

If you are a seasoned software/systems engineer or administrator, there are many things in this book that will challenge you. You might think you know it all. Well, you don't!

Conventions Used in This Book

The following typographical conventions are used in this book:

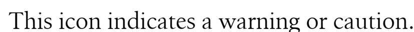
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

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

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

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

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


This icon indicates a warning or caution.

www.it-ebooks.info

Using Code Examples

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Amazon EC2* by Jurg van Vliet and Flavia Paganelli. Copyright 2011 I-MO BV, 978-1-449-39368-7."

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

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

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

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

www.it-ebooks.info

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9781449393687

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

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

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

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

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

Acknowledgments

There are many people we would like to thank for making this book into what it is now. But first of all, it would never have been possible without our parents, Aurora Gómez, Hans van Vliet, Marry van Vliet, and Ricardo Paganelli.

Right from the start we have been testing our ideas with many friends and colleagues; their early feedback shaped this book significantly. Thanks to Adam Dorell, Arjan van Woensel, Björn van Vliet, Dirk Groten, Eduardo Röhr, Eric Hammond, Federico Mikaelian, Fleur van Vliet, Grant Wilson, Indraneel Bommisetty, Joerg Seibel, Khalil Seyedmehdi, Marten Mickos, Matt Miesnieks, Pambo Pascalides, Pedro Moranga Gonçalves, Pim Derneden, Roy Chandra, Steven van Wel, Werner Vogels, Wouter Broekhof, and Zoran Kovačević.

Of course, you need "strange eyes" going over every detail and meticulously trying out examples to find errors. Our technical reviewers, Anthony Maës, Ben Immanuel, Graziano Obertelli, and Menno van der Sman, did just that.

And finally, there is the wonderful and extremely professional team at O'Reilly. Without Mike, Julie, and all the others there wouldn't even have been a book. To Amy Thomson, Adam Zaremba, Julie Steele, Mike Loukides, Sumita Mukherji, and the rest we met and worked with, thank you!

www.it-ebooks.info

CHAPTER 1
Introducing AWS

From 0 to AWS

By the late 1990s, Amazon had proven its success—it showed that people were willing to shop online. Amazon generated $15.7 million in sales in 1996, its first full fiscal year. Just three years later, Amazon saw $1.6 billion in sales, and Jeff Bezos was chosen Person of the Year by Time magazine. Realizing its sales volume was only 0.5% that of Wal-Mart, Amazon set some new business goals. One of these goals was to change from shop to platform.

At this time, Amazon was struggling with its infrastructure. It was a classic monolithic system, which was very difficult to scale, and Amazon wanted to open it up to third-party developers. In 2002, Amazon created the initial AWS, an interface to programmatically access Amazon's features. This first set of APIs is described in the wonderful book *Amazon Hacks* by Paul Bausch (O'Reilly), which still sits prominently on one of our shelves.

But the main problem persisted—the size of the Amazon website was just too big for conventional (web) application development techniques. Somehow, Jeff Bezos found Werner Vogels (now CTO of Amazon) and lured him to Amazon in 2004 to help fix these problems. And this is when it started for the rest of us. The problem of size was addressed, and slowly AWS transformed from "shop API" to an "infrastructure cloud." To illustrate exactly what AWS can do for you, we want to take you through the last six years of AWS evolution (see Figure 1-1 for a timeline). This is not just a historical journey, but also a friendly way to introduce the most important components for starting with AWS.

AWS has two unique qualities:

- It doesn't cost much to get started. For example, you don't have to buy a server to run it.
- It scales and continues to run at a low cost. For example, you can scale elastically, only paying for what you need.

www.it-ebooks.info

The second quality is by design, since dealing with scale was the initial problem AWS was designed to address. The first quality is somewhat of a bonus, but Amazon has really used this quality to its (and our) advantage. No service in AWS is useless, so let's go through them in the order they were introduced, and try to understand what problems they were designed to solve.

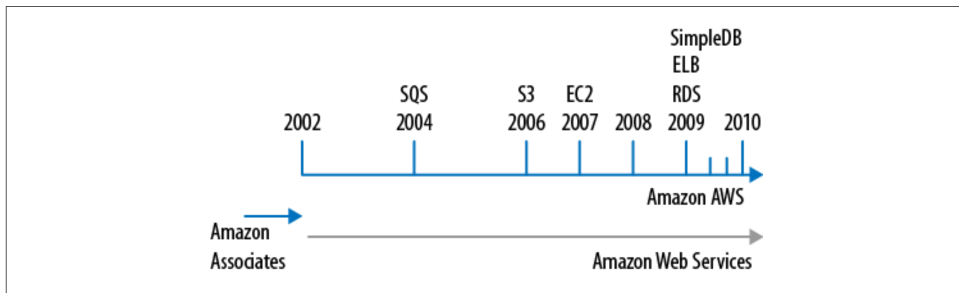


Figure 1-1. Timeline of AWS

Biggest Problem First

If your system gets too big, the easiest (and perhaps only) solution is to break it up into smaller pieces that have as few dependencies on each other as possible. This is often referred to as *decoupling*. The first big systems that applied this technique were not web applications; they were applications for big corporations like airlines and banks. These applications were built using tools such as CORBA and the concept of "component-based software engineering." Similar design principles were used to coin the more recent term service-oriented architecture or SOA which is mostly applied to web applications and their interactions.

Amazon adopted one of the elements of these broker systems, namely *message passing*. If you break up a big system into smaller components, they probably still need to exchange some information. They can pass messages to each other, and the order in which these messages are passed is often important. The simplest way of organizing a message passing system, respecting order, is a queue (Figure 1-2). And that is exactly what Amazon built first in 2004: Amazon Simple Queue Service or SQS.

By using SQS, according to AWS, "developers can simply move data between distributed components of their applications that perform different tasks, without losing messages or requiring each component to be always available." This is exactly what Amazon needed to start deconstructing its own monolithic application. One interesting feature of SQS is that you can rely on the queue as a buffer between your components, implementing *elasticity*. In many cases, your web shop will have huge peaks, generating 80% of the orders in 20% of the time. You can have a component that processes these orders, and a queue containing them. Your web application puts orders in the queue,

www.it-ebooks.info

and then your processing component can work on the orders the entire day without overloading your web application.

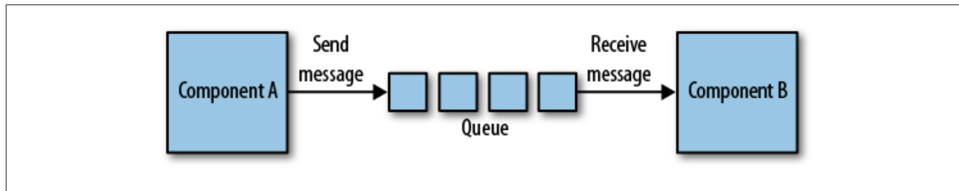


Figure 1-2. Passing messages using a queue

Infinite Storage

In every application, storage is an issue. There is a very famous quote attributed to Bill Gates that 640 K "ought to be enough for anybody." Of course, he denies having said this, but it does hit a nerve. We all buy hard disks believing they will be more than enough for our requirements, but within two years we already need more. It seems there is always something to store and there is never enough space to store it. What we need is infinite storage.

To fix this problem once and for all, Amazon introduced Amazon Simple Storage Service or S3. It was released in 2006, two years after Amazon announced SQS. The time Amazon took to release it shows that storage is not an easy problem to solve. S3 allows you to store objects of up to 5 terabytes, and the number of objects you can store is unlimited. An average DivX is somewhere between 600 and 700 megabytes. Building a video rental service on top of S3 is not such a bad idea, as Netflix realized.

According to AWS, S3 is "designed to provide 99.999999999% durability and 99.99% availability of objects over a given year." This is a bit abstract, and people often ask us what it means. We have tried to calculate it ourselves, but the tech reviewers did not agree with our math skills. So this is the perfect opportunity to quote someone else. According to Amazon Evangelist Jeff Barr, this many 9s means that, "If you store 10,000 objects with us, on average we may lose one of them every 10 million years or so." Impressive! S3 as a service is covered by a service level agreement (SLA), making these numbers not just a promise but a full contract.

S3 was extremely well received. Even Microsoft was (or is) one of the customers using S3 as a storage solution, as advertised in one of the announcements of AWS: "Global enterprises like Microsoft are using Amazon S3 to dramatically reduce their storage costs without compromising scale or reliability". In only two years, S3 grew to store 10 billion objects. In early 2010, AWS reported to store 102 billion objects in S3. Figure 1-3 illustrates the growth of S3 since its release.

www.it-ebooks.info

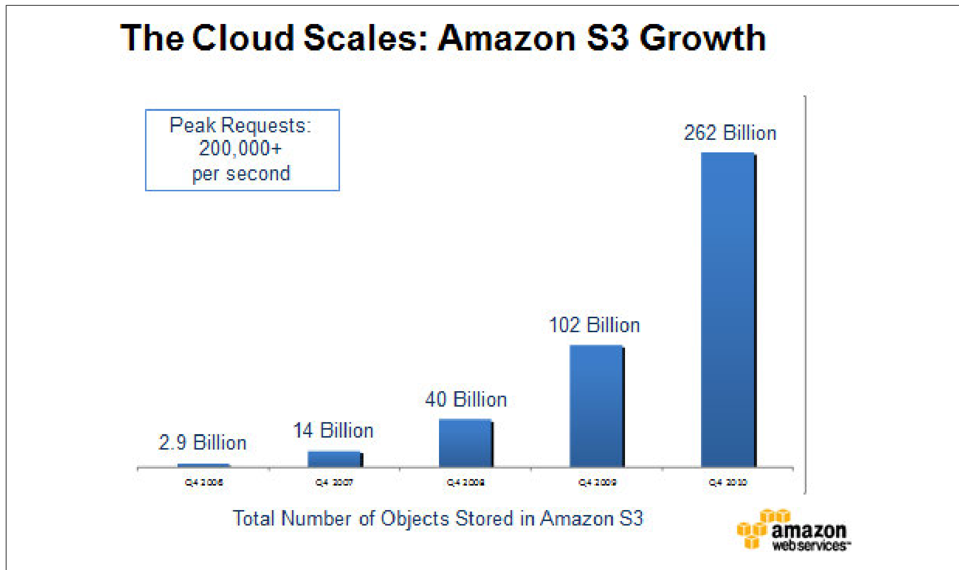


Figure 1-3. S3's huge popularity expressed in objects stored

Computing Per Hour

Though we still think that S3 is the most revolutionary of services because no one had solved the problem of unlimited storage before, the service with the most impact is undoubtedly Amazon Elastic Compute Cloud or EC2. Introduced as limited beta in the same year that S3 was launched (2006), EC2 turned computing upside down. AWS used XEN virtualization to create a whole new cloud category, Infrastructure as a Service, long before people started googling for IaaS. Though server virtualization already existed for quite a while, buying one hour of computing power in the form of a Linux (and later Windows) server did not exist yet.

Remember, Amazon was trying to decouple, to separate its huge system into components. For Amazon, EC2 was the logical missing piece of the puzzle because Amazon was in the middle of implementing a strict form of SOA. In Amazon's view, it was necessary to change the organization. Each team would be in charge of a functional part of the application, like wish lists or search. Amazon wanted each (small) team not only to build its own infrastructure, but also for developers to operate their apps themselves. Werner Vogels said it in very simple terms: "You build it, you run it."

www.it-ebooks.info

In 2007, EC2 was opened to everyone, but it took more than a year before AWS announced general availability, including SLA. There were some very important features added in the meantime, most of them as a result of working with the initial community of EC2 users. During this period of refining EC2, AWS earned the respect of the development community. It showed that Amazon listened and, more importantly, cared. And this is still true today. The Amazon support forum is perhaps its strongest asset.

By offering computing capacity per hour, AWS created elasticity of infrastructures from the point of view of the application developer (which is also our point of view.) When it was this easy to launch servers, which Amazon calls *instances*, a whole new range of applications became reachable to a lot of people. Event-driven websites, for example, can scale up just before and during the event and can run at low capacity the rest of the time. Also, computational-intensive applications, such as weather forecasting, are much easier and cheaper to build. Renting one instance for 10,000 hours is just as cheap as renting 10,000 instances for an hour.

Very Scalable Data Store

Amazon's big system is decoupled with the use of SQS and S3. Components can communicate effectively using queues and can share large amounts of data using S3. But these services are not sufficient as glue between the different applications. In fact, most of the interesting data is structured and is stored in shared databases. It is the relational database that dominates this space, but relational databases are not terribly good at scaling, at least for commodity hardware components. Amazon introduced Relational Database Server (RDS) recently, sort of "relational database as a service," but its own problem dictated that it needed something else first.

Although normalizing data is what we have been taught, it is not the only way of handling information. It is surprising what you can achieve when you limit yourself to a searchable list of structured records. You will lose some speed on each individual transaction because you have to do more operations, but you gain infinite scalability. You will be able to do many more simultaneous transactions. Amazon implemented this in an internal system called Dynamo, and later, AWS launched Amazon SimpleDB.

It might appear that the lack of joins severely limits the usefulness of a database, especially when you have a client-server architecture with dumb terminals and a mainframe server. You don't want to ask the mainframe seven questions when one would be enough. A browser is far from a dumb client, though. It is optimized to request multiple sources at the same time. Now, with a service specially designed for many parallel searches, we have a lot of possibilities. By accessing a user's client ID, we can get her wish list, her shopping card, and her recent searches, all at the same time.

www.it-ebooks.info

There are alternatives to SimpleDB, and some are more relational than others. And with the emergence of big data, this field, also referred to as NoSQL, is getting a lot of attention. But there are a couple of reasons why it will take time before SimpleDB and others will become successful. The most important reason is that we have not been taught to think without relations. Another reason is that most frameworks imply a relational database for their models. But SimpleDB is incredibly powerful. It will take time, but slowly but SimpleDB will surely find its place in (web) development.

Optimizing Even More

The core principle of AWS is optimization, measured in hardware utilization. From the point of view of a cloud provider like AWS, you need economies of scale. As a developer or cloud consumer, you need tools to operate these infrastructure services. By listening to its users and talking to prospective customers, AWS realized this very point. And almost all the services introduced in this last phase are meant to help developers optimize their applications.

One of the steps of optimization is creating a service to take over the work of a certain task. An example we have seen before is S3, which offers storage as a service. A common task in web (or Internet) environments is load balancing. And just as with storage or queues, it would be nice to have something that can scale more or less infinitely. AWS introduced a service called Elastic Load Balancing or ELB to do exactly this.

When the workload is too much for one instance, you can start some more. Often, but not always, such a group of instances doing the same kind of work is behind an Elastic Load Balancer (also called an ELB). To manage a group like this, AWS introduced Auto Scaling. With Auto Scaling you can define rules for growing and shrinking a group of instances. You can automatically launch a number of new instances when CPU utilization or network traffic exceeds certain thresholds, and scale down again on other triggers.

To optimize use, you need to know what is going on; you need to know how the infrastructure assets are being used. AWS introduced CloudWatch to monitor many aspects of the infrastructure assets. With CloudWatch, it is possible to measure metrics like CPU utilization, network IO, and disk IO over different dimensions like an instance or even all instances in one region.

AWS is constantly looking to optimize from the point of view of application development. It tries to make building web apps as easy as possible. In 2009, it created RDS, a managed MySQL service, which eases the burden of optimization, backups, scaling, etc. Early in 2010, AWS introduced the high availability version of RDS. AWS also complemented S3 with CloudFront, a very cheap content delivery network, or CDN. CloudFront now supports downloads and streaming and has many edge locations around the world.

www.it-ebooks.info

Going Global

AWS first launched on the east coast of the United States, in northern Virginia. From the start, the regions were designed with the possibility of failure in mind. A region consists of *availability zones*, which are physically separate data centers. Zones are designed to be independent, so failure in one doesn't affect the others. When you can, use this feature of AWS, because it can harden your application.

While AWS was adding zones to the US East region, it also started building new regions. The second to come online was Europe, in Ireland. And after that, AWS opened another region in the US, on the west coast in northern California. One highly anticipated new region was expected (and hinted at) in Asia Pacific. And in April 2010, AWS opened region number four in Singapore.

Growing into Your Application

In 2001, the Agile Manifesto for software development was formulated because a group of people felt it was necessary to have more lightweight software development methodologies than were in use at that time. Though this movement has found its place in many different situations, it can be argued that the Web was a major factor in its widespread adoption. Application development for the Web has one major advantage over packaged software: in most cases it is distributed exactly once. Iterative development is much easier in such an environment.

Iterative (agile) infrastructure engineering is not really possible with physical hardware. There is always a significant hardware investment, which almost always results in scarcity of these resources. More often than not, it is just impossible to take out a couple of servers to redesign and rebuild a critical part of your infrastructure. With AWS, you can easily build your new application server, redirect production traffic when you are ready, and *terminate* the old servers. For just a few dollars, you can upgrade your production environment without the usual stress.

This particular advantage of clouds over physical hardware is important. It allows for applying an agile way of working to infrastructures, and lets you iteratively grow into your application. You can use this to create room for mistakes, which are made everywhere. It also allows for stress testing your infrastructure and scaling out to run tens or even hundreds of servers. And, as we did in the early days of Layar, you can move your entire infrastructure from the United States to Europe in just a day.

In the following sections, we will look at the AWS services you can expect to use in the different iterations of your application.

Start with Realistic Expectations

When asking the question, "Does the application have to be highly available?", the answer is usually a clear and loud "yes." This is often expensive, but the expectation is

www.it-ebooks.info

set and we work very hard to live up to it. If you ask the slightly different question, "Is it acceptable to risk small periods of downtime provided we can restore quickly without significant loss of data?", the answer is the same, especially when it becomes clear that this is much less expensive. Restoring quickly without significant loss of data is difficult with hardware, because you don't always have spare systems readily available. With AWS, however, you have all the spare resources you want. Later, we'll show you how to install the necessary command-line tools, but all you need to start five servers is:

```
$ ec2-run-instances ami-480df921 -n 5
```

When it is necessary to handle more traffic, you can add servers—called *instances* in EC2—to relieve the load on the existing infrastructure. After adjusting the application so it can handle this changing infrastructure, you can have any number of instances doing the same work. This way of scaling—*scaling out*—offers an interesting opportunity. By creating more instances doing the same work, you just made that part of your infrastructure highly available. Not only is your system able to handle more traffic or more load, it is also more resilient. One failure will no longer bring down your app.

After a certain amount of scaling out, this method won't work anymore. Your application is probably becoming too complex to manage. It is time for something else; the application needs to be broken up into smaller, interoperating applications. Luckily, the system is agile and we can isolate and extract one component at a time. This has significant consequences for the application. The application needs to implement ways for its different parts to communicate and share information. By using the AWS services, the quality of the application only gets better. Now entire components can fail and the app itself will remain functional, or at least responsive.

Simply Small

AWS has many useful and necessary tools to help you design for failure. You can assign Elastic IP addresses to an instance, so if the instance dies or you replace it, you reassign the Elastic IP address. You can also use Elastic Block Store (EBS) volumes for instance storage. With EBS, you can "carry around" your disks from instance to instance. By making regular *snapshots* of the EBS volumes, you have an easy way to back up your data. An instance is launched from an *image*, a read-only copy of the initial state of your instance. For example, you can create an image containing the Ubuntu operating system with Apache web server, PHP, and your web application installed. And a boot script can automatically attach volumes and assign IP addresses. Using these tools will allow you to instantly launch a fresh copy of your application within minutes.

Most applications start with some sort of database, and the most popular database is MySQL. The AWS RDS offers MySQL as a service. RDS offers numerous advantages like backup/restore and scalability. The advantages it brings are significant. If you don't use this service, make sure you have an extremely good reason not to. Scaling a relational database is notoriously hard, as is making it resilient to failure. With RDS, you can start small, and if your traffic grows you can scale up the database as an immediate

www.it-ebooks.info

solution. That gives you time to implement optimizations to get the most out of the database, after which you can scale it down again. This is simple and convenient: priceless. The command-line tools make it easy to launch a very powerful database:

```
$ rds-create-db-instance kulitzer \
        --db-instance-class db.m1.small \
        --engine MySQL5.1 \
        --allocated-storage 5 \
        --db-security-groups default \
        --master-user-password Sdg_5hh \
        --master-username arjan \
        --backup-retention-period 2
```

Having the freedom to fail (occasionally, of course) also offers another opportunity: you can start searching for the boundaries of the application's performance. Experiencing difficulties because of increasing traffic helps you get to know the different components and optimize them. If you limit yourself in infrastructure assets, you are forced to optimize to get the most out of your infrastructure. Because the infrastructure is not so big yet, it is easier to understand and identify the problem, making it easier to improve. Also, use your freedom to play around. Stop your instance or scale your RDS instance. Learn the behavior of the tools and technologies you are deploying. This approach will pay back later on, when your app gets critical and you need more resources to do the work.

One straightforward way to optimize your infrastructure is to offload the "dumb" tasks. Most modern frameworks have facilities for working with media or static subdomains. The idea is that you can use extremely fast web servers or caches to serve out this static content. The actual dynamics are taken care of by a web server like Apache, for example. We are fortunate to be able to use CloudFront. Put your static assets in an S3 *bucket* and expose them using a CloudFront *distribution*. The advantage is that you are using a full-featured content delivery network with edge locations all over the world. But you have to take into account that a CDN caches aggressively, so change will take some time to propagate. You can solve this by implementing invalidation, building in some sort of versioning on your assets, or just having a bit of patience.

Growing Up

The initial setup is static. But later on, when traffic or load is picking up, you need to start implementing an infrastructure that can scale. With AWS, the biggest advantage you have is that you can create an elastic infrastructure, one that scales up and down depending on demand. Though this is a feature many people want, and some even expect out of the box, it is not applicable to all parts of your infrastructure. A relational database, for example, does not easily scale up and down automatically. Work that can be distributed to identical and independent instances is extremely well suited to an elastic setup. Luckily, web traffic fits this pattern, especially when you have a lot of it.

www.it-ebooks.info

Let's start with the hard parts of our infrastructure. First is the relational database. We started out with an RDS instance, which we said is easily scalable. It is, but, unaided, you will reach its limits relatively quickly. Relational data needs assistance to be fast when the load gets high. The obvious choice for optimization is caching, for which there are solutions like Memcached. But RDS is priceless if you want to scale. With minimum downtime, you can scale from what you have to something larger (or smaller):

```
$ rds-modify-db-instance kulitzer \
        --db-instance-class db.m1.xlarge \
        --apply-immediately
```

We have a strategy to get the most out of a MySQL-based data store, so now it is time to set up an elastic fleet of EC2 instances, scaling up and down on demand. AWS has two services designed to take most of the work out of your hands:

- Amazon ELB
- Amazon Auto Scaling

ELB is, for practical reasons, infinitely scalable, and works closely with EC2. It balances the load by distributing it to all the instances behind the load balancer. The introduction of *sticky sessions* (sending all requests from a client session to the same server) is recent, but with that added, ELB is feature-complete. With Auto Scaling, you can set up an autoscaling *group* to manage a certain group of instances. The autoscaling group launches and terminates instances depending on triggers, for example on percentage of CPU utilization. You can also set up the autoscaling group to add and remove these instances from the load balancer. All you need is an image that launches into an instance that can independently handle traffic it gets from the load balancer.

ELB's scalability comes at a cost. The management overhead of this scaling adds latency to the transactions. But in the end, human labor is more expensive, and client performance does not necessarily need ultra low latencies in most cases. Using ELB and Auto Scaling has many advantages, but if necessary, you can build your own load balancers and autoscaling mechanism. All the AWS services are exposed as APIs. You can write a daemon that uses CloudWatch to implement triggers that launch/terminate instances.

Moving Out

The most expensive part of the infrastructure is the relational database component. None of the assets involved here scales easily, let alone automatically. The most expensive operation is the join. We already minimized the use of joins by caching objects, but that is not enough. All the big boys and girls try to get rid of their joins altogether. Google has BigTable and Amazon has SimpleDB, both of which are part of what is now known as NoSQL. Other examples of NoSQL databases are MongoDB and Cassandra, and they have the same underlying principle of not joining.

www.it-ebooks.info

The most radical form of minimizing the use of joins is to decouple, and a great way to decouple is to use queues. Two applications performing subtasks previously performed by one application are likely to need less severe joins. Internally, Amazon has implemented an effective organization principle to enforce this behavior. Amazon reorganized along the lines of the functional components. Teams are responsible for everything concerning their particular applications. These decoupled applications communicate using Amazon SQS and Amazon Simple Notification Service (SNS), and they share using Amazon SimpleDB and Amazon S3.

These teams probably use MySQL and Memcached and ELB to build their applications. But the size of each component is reduced, and the traffic/load on each is now manageable once more. This pattern can be repeated again and again, of course.

"You Build It, You Run It"

Perhaps by chance, but probably by design, AWS empowers development teams to become truly agile. It does this in two ways:

- Getting rid of the long-term aspect of the application infrastructure (investment)
- Building tools to help overcome the short-term aspect of operating the application infrastructure (failure)

There is no need to distinguish between building and running, and according to Werner Vogels, it is much better than that:

> Giving developers operational responsibilities has greatly enhanced the quality of the services, both from a customer and a technology point of view. The traditional model is that you take your software to the wall that separates development and operations, and throw it over and then forget about it. Not at Amazon. You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service.

This lesson is interesting, but this particular change in an organization is not always easy to implement. It helped that Vogels was the boss, though it must have cost him many hours, days, and weeks to convince his colleagues. If you are not the boss, it is even more difficult, though not impossible. As we have seen before, AWS offers ways to be agile with infrastructures. You can tear down servers, launch new ones, reinstall software, and undo entire server upgrades, all in moments.

Individuals and Interactions: One Team

In bigger organizations, there is an IT department. Communication between the organization and its IT department can difficult or even entirely lacking. The whole activity of operating applications can be surrounded with frustration, and everyone feels powerless. Smaller companies often have a hosting provider, which can be very similar

www.it-ebooks.info

to an IT department. A hosting provider tends to be a bit better than an IT department because you can always threaten to replace it. But the lock-in is significant enough to ignore these issues; for a small company it is generally more important to focus on development than to spend time and energy on switching hosting providers.

Let's start with one side: the IT department or hosting provider. Its responsibility is often enormous. IT department members have to make decisions on long-term investments with pricetags that exceed most product development budgets. These investments can become difficult projects with a huge impact on users. At the same time, the IT department has to make sure everything runs fine 24/7. It is in a continuous battle between dealing with ultra long term and ultra short term; there seems to be nothing in between.

Now for the development team. The work of the development team is exactly in between the long term and the short term. The team is asked to deliver in terms of weeks and months, and often makes changes in terms of days. During the development and testing phases, bugs and other problems are part of the process, part of the team's life. But once in production, the application is out of the team's hands, whether they like it or not.

Organizations can handle these dynamics by creating complex processes and tools. Because each group typically has no understanding of the other's responsibilities, they tend to formalize the collaboration/communication between the teams, making it impersonal. But as the Agile Manifesto states, in developing software *individuals and interactions* are more valuable than processes and tools. With AWS, the investment part of infrastructures is nonexistent. And AWS helps you manage the ultra short term by providing the tools to recover from failure. With AWS, you can *merge the responsibility of running the application with the responsibility of building it*. And by doing this, you turn the focus on the people and their interactions instead of on creating impersonal and bureaucratic processes.

Working Software: Shared Responsibility

Deploying software means moving the application from development to the "other side," called *production*. Of course, the other side—the IT department in the traditional structure—has already committed to a particular SLA. As soon as the application is moved, the IT department is on its own. As a consequence, members want or need to know everything necessary to run the application, and they require documentation to do so.

This documentation is an SLA itself. If there is a problem related to the software that is not included in the documentation, fingers will point to the development team. The documentation becomes a full description of every aspect of the application, for fear of liability.

www.it-ebooks.info

But in the end, there is only one thing that matters, and that is whether the application is running. This is not very difficult to determine if the responsibility is shared; the team members will quickly discuss a solution instead of discussing who is to blame. So the thing to do is to build working software together, as a team. Remove the SLAs and merge the functions of two teams into one. When something doesn't work, it needs to be fixed—it does not always have to be debated first. Documentation in this context becomes less important as a contract between parts, and becomes an aid to keep the application running.

Customer Collaboration: Evolve Your Infrastructure

Wherever IT is present, there is an SLA. The SLA is regarded as a tool in managing the process of IT infrastructure, where the bottom line is the number of nines. In reality it is a tool designed to facilitate cooperation, but is often misused for the purpose of deciding who is responsible for problems: development or operations.

It can be difficult to negotiate this contract at the time of application development. There is a huge difference between "we need to store audio clips for thousands of customers" and "storage requirements are estimated to grow exponentially from 500 GB to 5 TB in three years." The problem is not so much technical as it is that expectations (dreams, often) are turned into contract clauses.

You can change contract negotiation into *customer collaboration*. All you need to do is merge the two responsibilities: building and running the application becomes a shared challenge, and success is the result of a shared effort. Of course, in this particular example it helps to have Amazon S3, but the point is that requirements change, and collaboration with the customer is better suited for handling those changes than complex contract negotiations.

Responding to Change: Saying Yes with a Smile

At the end of the project, just two weeks before launch, the CEO is shown a sneak preview of the new audio clip platform. She is very excited, and proud of the team effort. The meeting is positive and she is reassured everything is planned for. Even if the dreams of millions of customers come true, the platform will not succumb to its success because it's ready to handle a huge number of users.

In the evening, she is telling her boyfriend about her day. She shares her excitement and they both start to anticipate how they would use the new platform themselves. At some point he says, "Wouldn't it be great to have the same platform for video clips?" Of course, he doesn't know that this whole project was based on a precondition of audio-only; neither does the CEO.

In the morning, she calls the project manager and explains her idea. She is still full of energy and says enthusiastically, "the functionality is 100% perfect, only we want audio *and* video." The project manager knows about the precondition, and he also knows

www.it-ebooks.info

that video files are significantly bigger than audio files. However, the CEO doesn't want to hear buts and objections about moving away from the plan; she wants this product to change before launch.

In Short

In this chapter we walked you through the recent history of AWS. We showed how each of the AWS services was created to solve a particular problem with Amazon's platform. We gave you a brief overview of the different AWS services you can use to build, monitor, and scale your cloud infrastructure. And finally, we talked about how developing with AWS is naturally agile and allows you to make the infrastructure building and running part of the development process.

In the rest of the book, we'll show how all these services actually work, so get ready to stop reading and start doing! In the next chapter, we will start with migrating a simple web application to AWS using EC2, RDS, S3, and CloudFront.

www.it-ebooks.info

CHAPTER 2

Starting with EC2, RDS, and S3/CloudFront

So far we have talked about AWS. You have seen where it comes from, how it can help you with growing your application, and how a virtual infrastructure on AWS benefits your work. It is important to understand the context, because it helps you select the services and tools you need to move your app to AWS. But real experience only comes with practice!

So let's get down to business. First off, you need an AWS account, which requires a valid credit card and a phone. Once you have an account, all AWS services are at your disposal. To use the services, you need to set up your local environment for working with the command-line tools in case you need them. For now, the AWS Console and command-line tools are enough, but there are commercial and noncommercial applications and services available that offer something extra. Last, but not least, you might want to monitor your application and get tools to fix it on the move as well.

With the account activated and the tools available, all we need is an application to build. Working on a real application is more fun than just a demo, so we'll use one of our applications in production, called Kulitzer. Kulitzer.com calls itself "the contest platform for creative people. You can join a contest, enter your work, and take a seat in the jury. Can't find any contest you like? Start your own!" Kulitzer is a Rails application, developed in New Zealand by Arjan van Woensel. Very early in the process, van Woensel decided he wanted Kulitzer on AWS. The main reasons for this were price and scalability.

In this chapter, we will move Kulitzer.com to AWS. You can follow along with your own app; it doesn't have to be a Rails application. We will not be showing much code; rather, we will concentrate on the infrastructure and tools necessary to work with AWS. If you don't have an app at hand, there are many open source apps available. We'll build the app using Amazon RDS, a flavor of MySQL, because we love it. But you can just as easily run your own PostgreSQL database, for example. It's a good idea to follow

www.it-ebooks.info

along, and if you carefully stop your instances, it won't cost you more than what you paid for this book, since AWS offers a free tier.

Setting Up Your Environment

Before you can start setting up your instances and creating Amazon Machine Images (AMIs), you have to set up a good working environment. You don't need much, as AWS is 100% virtual. But you do need a couple of things:

- A desktop or laptop (with Internet access, of course)
- A credit card (for setting up an AWS account)
- A phone (to complete the registration process)

We use a MacBook, but many of our colleagues work on Ubuntu or Windows. We'll use the terminal app, and bash as our shell. The only real requirement is to have Java installed for the command-line tools. For the rest, any browser will do.

Your AWS Account

Creating an AWS account is pretty straightforward. Go to the AWS website and click the Sign Up Now button. You can use an existing Amazon account to sign up for AWS.

This will create the Amazon.com account you are going to use to access AWS. But this is not enough. To start with, you need EC2. On the AWS site, click on Amazon Elastic Compute Cloud under Projects. Click the Sign Up For Amazon EC2 button. With your credit card and phone ready, you can complete the signup; the last stage of this process is a validation step where you are called by Amazon (see Figure 2-1).

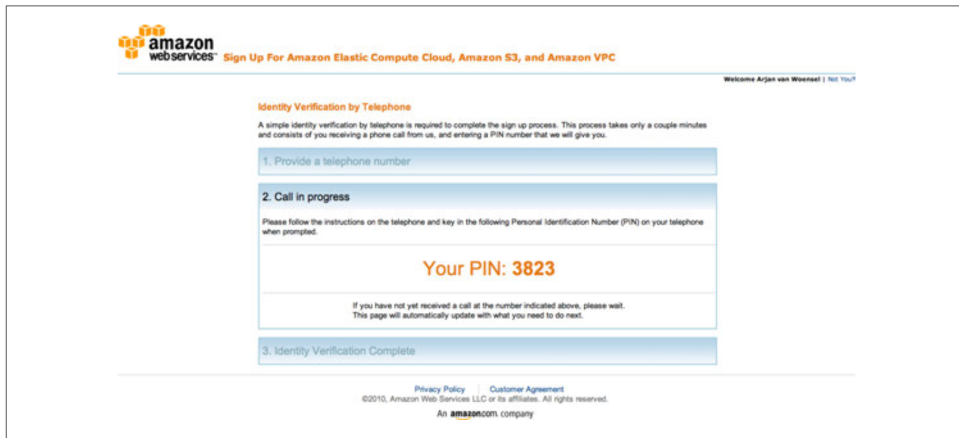


Figure 2-1. Identity verification

www.it-ebooks.info

Later, you'll need other services, but signing up for those is much easier after the credit card validation and account verification have been taken care of.

> You might be wondering how much all this will cost you. To start with, signing up for services doesn't cost anything. When you start actually using the services, Amazon provides a free usage tier for every service, which lasts for a year after signup. For example, it's possible to use a micro instance for free for 750 hours a month. For S3 storage, the limit is 5 GB. Also, services like SQS and SimpleDB offer some free space to play around with. After you have used up your free tier, a micro instance will cost you only about US$0.02 per hour. Remember to stop the instances when you are not using them, and you will have a lot of resources to use for experimenting.
>
> If you are wondering how much it will cost to have a real application in the Amazon cloud, take a look at the Amazon Simple Monthly Calculator. There, you can fill in how many servers you will use, how much storage, how many volumes, bandwidth, etc., and it will calculate the cost for you.

Command-Line Tools

Take a few minutes to install the command-line tools. Even though most EC2 functionality is supported by the web-based AWS Console, you will occasionally have to use the command line for some features that are not yet implemented in the Console. Plus, later on you might want to use the command-line tools to script tasks that you want to automate.

Running the command-line tools is not difficult if you set up the environment properly. Accessing AWS is safe; it is protected in a couple of different ways. There are three types of access credentials (you can find these in the Account section if you look for Security Credentials at *http://aws.amazon.com/*):

- *Access keys*, for REST and Query protocol requests
- *X.509 certificates*, to make secure SOAP protocol requests
- *Key pairs*, in two different flavors, for protecting CloudFront content and for accessing your EC2 instances

You will need X.509 credentials, the EC2 API tools, and the RDS Command Line Toolkit to follow along with the exercises in this chapter. Working with these tools requires you to specify your security credentials, but you can define them in environment variables. Our "virgin" AWS account does not have much, and it doesn't have X.509 certificates yet. You should still be on the Security Credentials page, where you can either upload your own or create them. We can ask AWS to create our X.509 certificates and immediately download both the access key ID and the secret access key (Figure 2-2).

www.it-ebooks.info

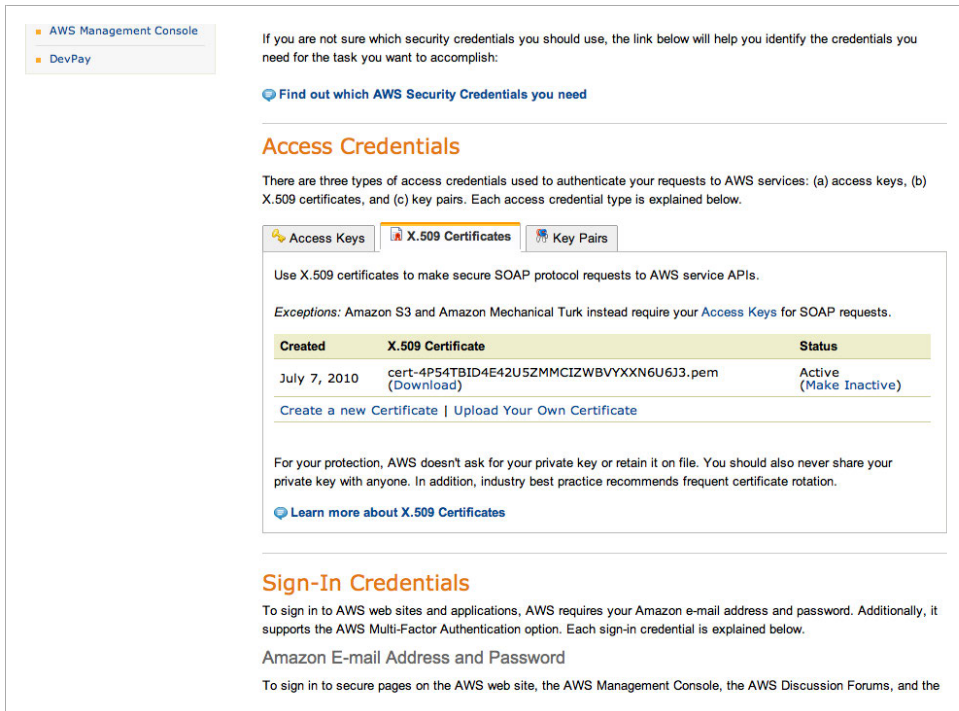


Figure 2-2. AWS credentials

With your downloaded certificates, you can set the environment variables. For this, create a bash script called `initaws` like the one listed below (for Windows, we would have created a BAT script). Replace the values of the variables with the locations of your Java home directory, EC2, and RDS command-line tools, the directory where you downloaded your key, and your secret keys:

```
#!/bin/bash
export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home
export EC2_HOME=/Users/arjan/src/ec2-api-tools-1.3-46266
export AWS_RDS_HOME=/Users/arjan/src/RDSCli-1.1.005
export PATH="$EC2_HOME/bin:$AWS_RDS_HOME/bin:$PATH"

export EC2_KEY_DIR=/Users/arjan/.ec2
export EC2_PRIVATE_KEY=${EC2_KEY_DIR}/pk-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_CERT=${EC2_KEY_DIR}/cert-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
```

You can now run the script in your terminal with `source initaws`. Let's see if this worked by invoking another command, `ec2-describe-regions`:

```
$ ec2-describe-regions
REGION   eu-west-1      ec2.eu-west-1.amazonaws.com
REGION   us-east-1      ec2.us-east-1.amazonaws.com
REGION   us-west-1      ec2.us-west-1.amazonaws.com
REGION   ap-southeast-1 ec2.ap-southeast-1.amazonaws.com
```


www.it-ebooks.info

We will soon discuss the concept of regions, but if you see a list similar to this, it means your tools are set up properly. Now you can do everything that AWS offers you. We'll take a look at the AWS Console next. The command-line tools offer something unique, though: we can easily create scripts to automate working with AWS. We will show that later on in this chapter.

Identity and Access Management

In this book, we mostly use the credentials of the account itself. This simplifies working with AWS, but it does not conform to industrial-grade security practices. Amazon realized that and introduced yet another service, called Identity and Access Management (IAM). With IAM, you can, for example, create users with a very limited set of rights. We could create a user `s3` that can access only the S3 service on any S3 resources of our account:

```
$ iam-usercreate -u s3 -k
AKIAID67UECIAGHXX54A
py9RuAIfgKz6N1SYQbCc+bFLtE8C/RX12sqwGrIy

$ iam-useraddpolicy -u s3 -p S3_ACCESS \
    -e Allow -a "s3:*" -r "*"
```

When creating the user, the `-k` option indicates that a set of access keys should be created for this new user. You can create users for the sole purpose of making backups that have access only to SimpleDB and EBS snapshot creation, for example. Creating IAM users reduces the risk of a breach of your instances. This simple example does absolutely no justice to IAM; to learn more, visit the AWS Identity and Access Management (IAM) page on the AWS portal.

It is good to be aware of one particular AWS tool relating to IAM and security policies on AWS. Creating a policy can be somewhat overwhelming; trying to figure out all the different privileges and service names requires a lot of patience and determination. But just recently, AWS introduced the AWS Policy Generator. With this online tool, you can easily generate policies to be added to S3, EC2, or any of the other available AWS services.

AWS Management Console

What is there to say about the AWS Management Console? We have been using it ever since it was launched. There are some changes we would like to see, but it is a very complete tool (Figure 2-3).

We can do most basic AWS tasks with the AWS Console. At the time of this writing, it offers Amazon S3, Amazon EC2, Amazon VPC, Amazon Elastic MapReduce, Amazon CloudFront, Amazon RDS, and Amazon SNS.

www.it-ebooks.info

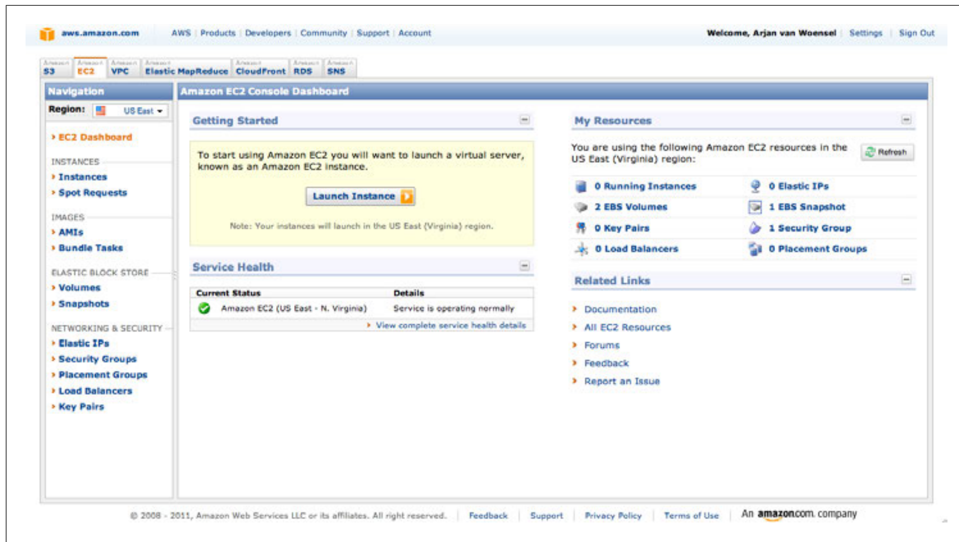


Figure 2-3. The AWS Console

Other Tools

With the AWS Console and the command-line tools, we have nearly everything we need. The only thing missing is something to help us monitor and fix problems when they arise. Because AWS is 100% virtual, you don't need to replace broken hardware anymore. It is not necessary to go anywhere to fix malfunctioning components of your infrastructure. It is even possible to fix the encountered problems from a smartphone.

We checked what's out there for managing Amazon clouds on a smartphone, but we didn't find tools that were sufficient for our needs. We wanted a single application to monitor and manage our infrastructure. With smartphone platforms like iOS from Apple, you can go a long way. But the limited functionality offered for background processes in iPhone sets a limit to what you can do with your application in terms of monitoring. Therefore, we chose Android to develop an application called Decaf. With Decaf, you can manage and monitor a virtual infrastructure built on AWS.

There are alternative monitoring applications and services we could use, but most are quite expensive. Often, alerting is part of a more extensive monitoring platform like Nagios or Cacti. We prefer to use Amazon CloudWatch, though, and the only thing we need is simple monitoring. In later chapters, we'll discuss CloudWatch in depth and show you how to operate your infrastructures.

www.it-ebooks.info

Choosing Your Geographic Location, Regions, and Availability Zones

With our tools set up, we can get to work. Let's start building the infrastructure for the Kulitzer application. First thing is choosing a region where our servers will live. At this moment, we can choose from the following:

- EU West—Ireland (`eu-west-1`)
- US East—Northern Virginia (`us-east-1`)
- US West—California (`us-west-1`)
- Asia Pacific—Singapore (`ap-southeast-1`)

Joerg Seibel and his team develop and maintain Kulitzer, while we (the authors) build the infrastructure in the Netherlands. But the regions Kulitzer will be targeting at the start are the United States and Europe.

US East, the first region to appear, is relatively close to the majority of customers in both the United States and Europe. The European region was launched soon afterward, making it possible to target these regions individually. In the case of Kulitzer, we had to make a choice because we have limited funds. US East is the default region, and slightly less expensive than the others, so it's the best option for us.

Every region has a number of *availability zones*. These zones are designed to be physically separated but still part of one data network. The purpose of different availability zones is to make your infrastructure more resilient to failures related to power and network outages. At this point, we will not be utilizing this feature, but we will have to choose the right availability zone in some cases.

If you work with AWS, it is good to know that the tools operate by default on the US East region. If you don't specify otherwise, everything you do will be in this region. There is no default availability zone, though. If you don't specify an availability zone when you create an instance, for example, AWS will choose one for you.

Choosing an Architecture

So, US East it is. What's next? We have a standard, three-tiered application, consisting of a database, an application server, and a web server. We have to build a small architecture, so we'll combine the application and web server. We do have the luxury of offloading the dumb traffic to Amazon CloudFront, a content distribution network we can utilize as a web server. The initial architecture will look like the one shown in Figure 2-4.

www.it-ebooks.info

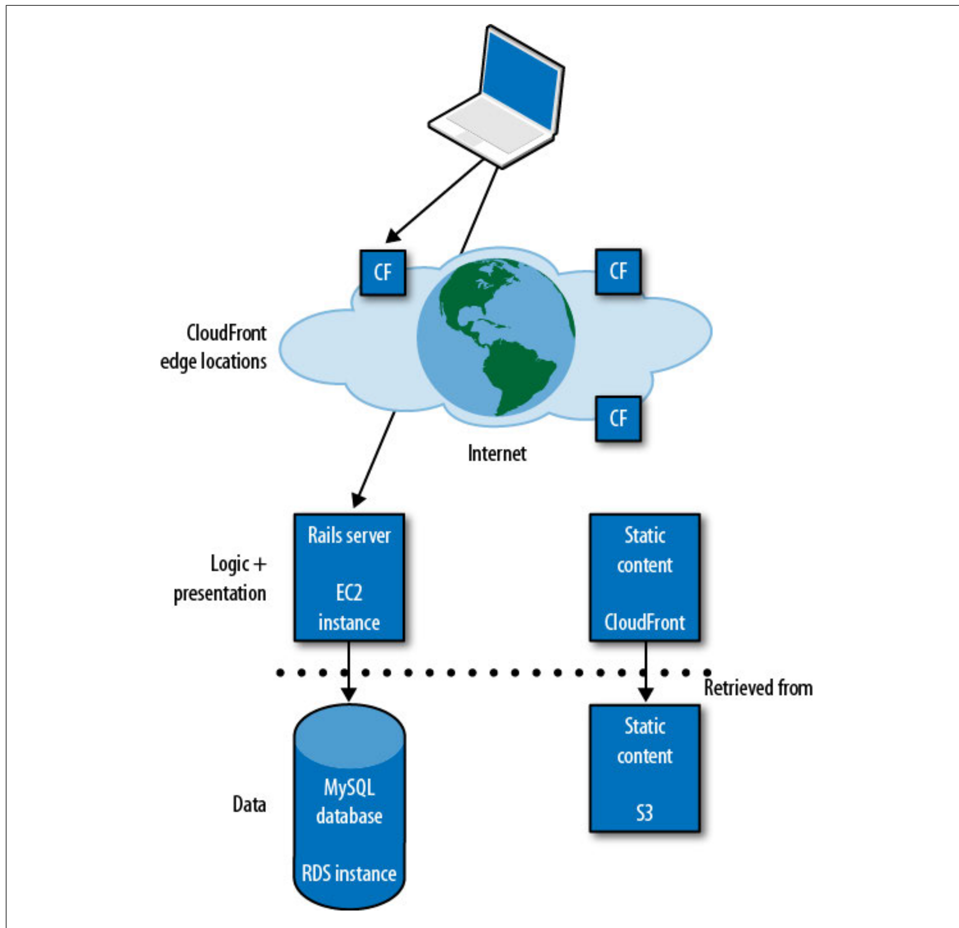


Figure 2-4. Kulitzer architecture v1.0

Our operating system of preference is Ubuntu, but you can run various other flavors of Linux, OpenSolaris, or Windows Server 2003/2008. Our Rails stack consists of Rails web framework 2.3.5, Apache web server, and Passenger to facilitate deployments. The server will also be sending mail, for which we use Postfix. The actual installation is beyond the scope of the book. We'll get you on the EC2 Ubuntu instance, where you can continue installing your application.

Creating the Rails Server on EC2

OK, here we go! Amazon Elastic Compute Cloud (EC2) is the heart of AWS. It consists of many different assets you need to understand before an EC2 instance (server) becomes operational. The different features will be introduced in the order in which they are needed.

www.it-ebooks.info

Creating a Key Pair

A key pair is one of the ways AWS handles security. It is also the only way to get into your fresh instance the first time you launch it. You can create a Secure Shell (SSH) key pair and pass it on to the instance you launch. The public key will be stored in the instance in the right place, while you keep the private key to log in to your instance.

You can create a key pair through the AWS Console. Go to Key Pairs and click Create Key Pair. Give it a name and store the downloaded private key somewhere safe (Figure 2-5)—you won't be able to download it again.

You can also import your own existing SSH key pair to AWS using the `ec2-import-keypair` command, like in the following example:

```
ec2-import-keypair --region us-east-1 --public-key-file .ssh/id_rsa.pub arjan
```

where `arjan` is the name of the key pair. You have to import your key pair to each region where you will use it.

Remember when we set up the command-line tools? If you look at that script, you can see we created a directory for the other certificates. This is a good place to store the key pair too. It also gives you a way to organize your files if you work with multiple AWS accounts.

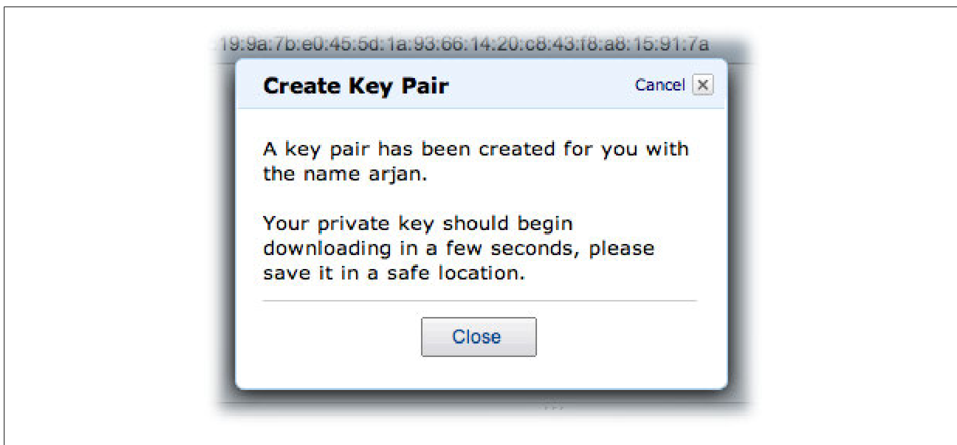


Figure 2-5. Create a key pair

Finding a Suitable AMI

An AMI is like a boot CD. It contains the root image with everything necessary to start an instance. There are many publicly available AMIs, and you can create your own preconfigured one for your needs. The available operating systems include various

www.it-ebooks.info

flavors of Linux, Windows, and OpenSolaris. Often, AMIs are simply referred to as images.

There are two different kinds of AMIs. The "old" kind of AMI is stored on S3. Launching an instance from an S3-backed AMI (as they are called) gives you an instance with the root device in the machine itself. This is a bit abstract, but remember that devices that are part of the instance itself are gone when the instance is gone. AWS uses the term *ephemeral storage* for these devices. This is also the reason why instances launched from an S3-backed AMI cannot be stopped and started; they can only be restarted or terminated.

The other, newer kind of AMI is stored in EBS. The most important difference for now is that the root device is not ephemeral anymore, but an EBS volume will be created that can survive the instance itself. Because of this, an EBS-backed instance can now be stopped and started, making it much easier to use the instance only when you need it. A stopped instance does not cost you anything apart from the EBS storage used. (EBS will be described in detail later.)

> EBS-backed AMIs are much more convenient than S3-backed AMIs. One of the drawbacks, though, is that these AMIs are not easily transferred. Because an S3-backed AMI is in S3 (obviously), you can copy it to wherever you want. This is problematic for EBS-backed AMIs. This poses a problem if you want to migrate your instances and/or AMIs to another region.

We are most familiar with Ubuntu. For Ubuntu, you can use the Canonical (derived) AMIs. The first good source for Ubuntu AMIs was Alestic. Though Ubuntu now builds and maintains its own EC2 AMIs, we still find ourselves going back to Alestic.

The AWS Console allows you to search through all available AMIs. It also tells if the AMI is EBS- or S3-backed; an *instance-store* root device type means S3-backed, and *ebs* means EBS-backed. For example, in Figure 2-6, we filtered for AMIs for Ubuntu 10.4, and two are listed: one of type *instance-store* and the other of type *ebs*. However, the Console does not really help you in determining the creator of the image, so be sure you are choosing exactly the image you intend to use by getting the AMI identifier from its provider (in our case, Alestic).

For now, there is one more important thing to know when you are looking for the right AMIs. Instances are either 32-bit or 64-bit. AMIs, obviously, follow this distinction. An AMI is either 32-bit or 64-bit, regardless of how it is backed. We want to start small, so we'll choose the 32-bit AMI for launching a small instance.

Setting Up the Web/Application Server

Before you actually select the AMI and click Launch, let's take a look at what an instance is. An *instance* is the virtual counterpart of a server. It is probably called an instance

www.it-ebooks.info

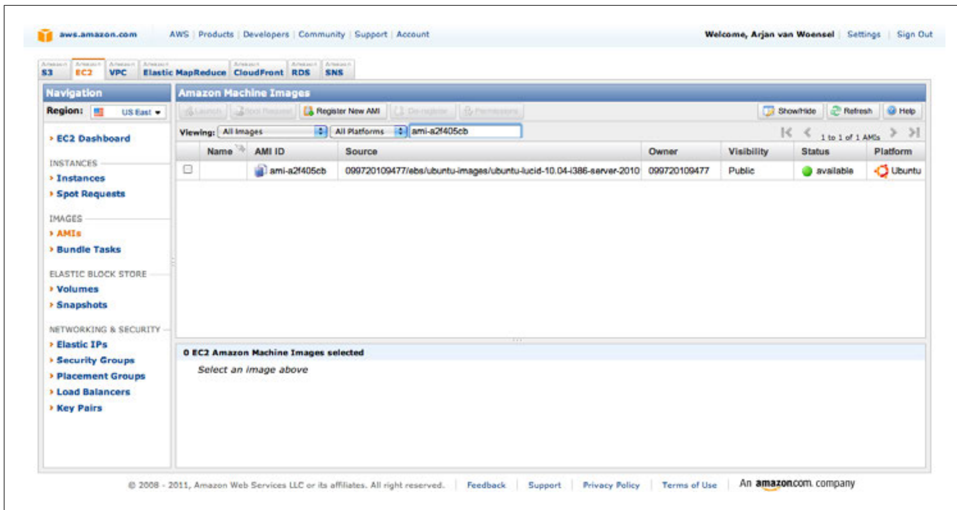


Figure 2-6. Find your AMIs

because it is launched from an immutable image. Instances come in types. You can think of a type as the size of the instance. The default type is *Small*, which is a 32-bit instance. The other 32-bit instances are *Micro* and *High-CPU Medium*. Micro instances support both 32- and 64-bit architecture, and all the others are all exclusively 64-bit instances. This is important because it shows you that scaling up (scaling by using a bigger server) is quite constrained for 32-bit instances. (You can launch any type of 64-bit instance from a 64-bit AMI.) A Micro instance costs approximately US$0.02 per hour. On the other end, a *Quadruple Extra Large* instance is available for approximately US$2.40 per hour.

According to the AWS documentation, an instance provides a "predictable amount of dedicated compute capacity." For example, the Quadruple Extra Large instance provides:

- 68.4 GB of memory
- 26 EC2 compute units (8 virtual cores with 3.25 EC2 compute units each)
- 1690 GB of instance storage
- 64-bit platform
- High I/O performance
- API name: m2.4xlarge

www.it-ebooks.info

AWS describes an EC2 compute unit like this: "One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early-2006 1.7 GHz Xeon processor referenced in our original documentation. Over time, we may add or substitute measures that go into the definition of an EC2 Compute Unit, if we find metrics that will give you a clearer picture of compute capacity."

Launching an instance (Request Instances Wizard)

The AWS Console guides you through the process of launching one or more instances with the Request Instances Wizard. We'll use this wizard to explain what it means to launch an instance. Step one is choosing the AMI. As we said, we use Ubuntu and the public i386 Ubuntu Lucid 10.04 AMI from Alestic.

Next, we have to determine the instance details. You can set the number of instances you want to launch; we only need one. You can explicitly choose your availability zone, but we'll let AWS choose a zone for now. And because this AMI is for 32-bit instances, we can choose between Micro, Small, and High-CPU Medium. You can either launch an instance or request a *spot instance*.

> For new users, there is a limit of 20 concurrent instances. If you need more than 20 instances, you request it from Amazon by filling out the Request to Increase Amazon EC2 Instance Limit form.

We'll leave the Advanced Instance Options on their default settings; the option of interest here is CloudWatch Monitoring, which we'll talk about later. All instances have basic monitoring enabled by default, but we can enable detailed monitoring once the instance is running.

In the following step, you can add tags to your instance or key-value pairs for the purpose of easily identifying and finding them, especially when you have a bigger infrastructure. The identifier of an instance is not very user-friendly, so this is a way to better organize your instances. This use of tags is also available for other EC2 components, such as images, volumes, etc.

Next, we can create a key pair or choose an existing one. We created our key pair before, so we'll select that one. This will allow us to access the new instance. When launched, the public key is added to the user called "ubuntu" (in our case), and we can log in with the private key we downloaded when creating the key pair. As we said before, we'll create our own images later on, including users and their public keys, so that we can launch without a key pair.

The last step before launching is to choose the *security groups* the instance will be part of. A security group defines a set of firewall rules or allowed connections, specifying who can access the instance and how. You can define who has access by using an IP

www.it-ebooks.info

address, IP range, or another security group. You specify how it can be accessed by specifying TCP, UDP, or ICMP in a port or range of ports. So, for example, there is a *default* security group provided by Amazon, which allows all network connections coming from the same default group.

Several instances can use the same security group, which defines a kind of profile. For example, in this case, we will define a security group for the web server, which we can then apply to all the web server instances if we scale out. Also, an instance can have several security groups, each adding some set of rules. Security groups are a very powerful way of specifying security restrictions. Once the system is running, you cannot add or remove instances from security groups. You can, however, change a security group by adding and/or removing new allowed connections.

We will create a security group that allows SSH, HTTP, and HTTPS connections from any IP address.

> The security groups screen in the wizard is called Configure Firewall. We tend to think of security groups as a combination of a firewall and VLANs (Virtual LANs). A security group is its own little network, where instances within a group can freely communicate without constraints. Groups can allow connections to other groups, again unconditionally. And you can selectively open a group to an outside address or group of addresses. It is straightforward to create DMZ (demilitarized zones) for a database or group of application servers.

With everything set up, we can review what we did and launch our instance (Figure 2-7).

Of course, we could have done it quicker using the command-line tools, though it would not have been so informative. But next time, you can do exactly the same thing with the following commands (you only have to create the key pair and security group once):

```
# create the key pair
$ ec2-add-keypair arjan

# create a security group called 'web'
$ ec2-add-group web -d 'All public facing web (port 80 and 443) instances'
$ ec2-authorize web -P tcp -p 22 -s 0.0.0.0/0
$ ec2-authorize web -P tcp -p 80 -s 0.0.0.0/0
$ ec2-authorize web -P tcp -p 443 -s 0.0.0.0/0

# launch an instance
$ ec2-run-instances ami-714ba518 \
        --instance-count 1 \
        --instance-type m1.small \
        --key arjan \
        --group web
```

Notice that the IP range is specified using Classless Inter-Domain Routing (CIDR).

www.it-ebooks.info

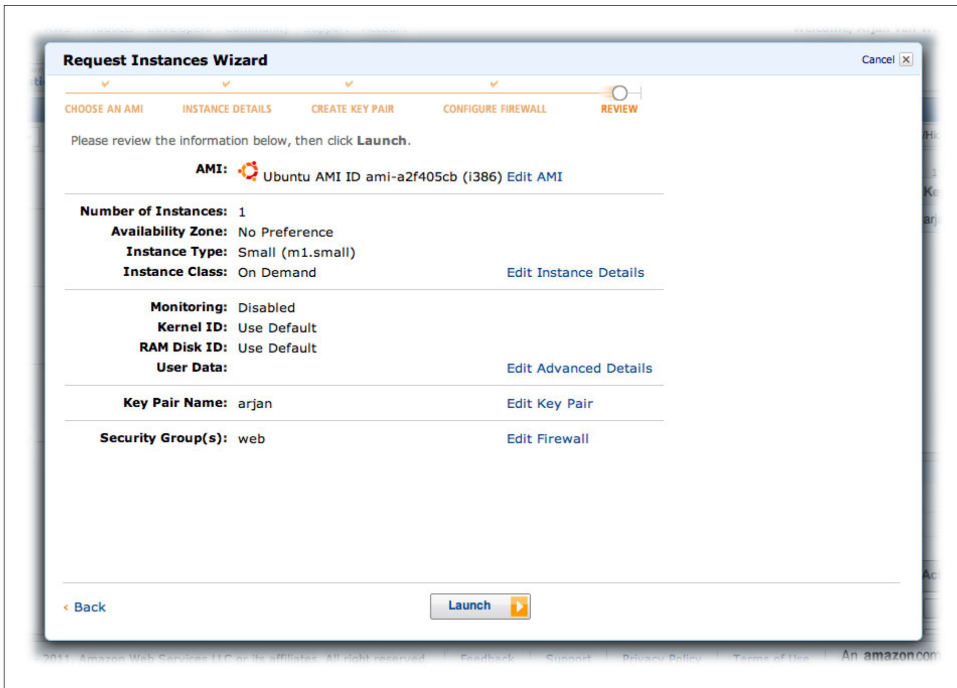


Figure 2-7. Ready to launch an instance

Setting up the instance

Our new instance is launching and shouldn't take very long. Once the instance is available, we are going to prepare it for Rails. (Installing Rails is beyond the scope of this book.) Preparing the instance for something like PHP or Django is not fundamentally different. Let's first try to log in to our new instance using the certificate of the key pair, the user ubuntu, and what is shown in the Console as Public DNS for the instance:

```
$ ssh -i ~/.ec2/arjan.pem ubuntu@ec2-184-72-128-63.compute-1.amazonaws.com
```

If you end up in a bash shell, logged in as ubuntu, you can claim success! You are now able to do everything without a password. Not really a nice idea, so we usually add at least one user to make sure we don't make it too easy for those who mean us harm.

Setting up the instance is, in principle, the same as setting up a physical server. There are a number of interesting differences, though, and we can use some of them to make life easier. Just like most physical servers, an EC2 instance comes with local disk storage. This storage lives as long as the instance lives. For EBS-backed instances, this means it persists when stopped, but vanishes when terminated. It also means it is gone when the instance unexpectedly dies.

To persist this local disk storage, we have two options: one way is to create an image based on the instance, and the other way is to use EBS volumes. An image is immutable, which means changes to the instance after the image has been created do not change

www.it-ebooks.info

the image. An EBS volume, however, is independent. You can attach an EBS volume to only one instance at a time. Another interesting feature of EBS volumes is that you can take incremental snapshots. We'll use this for our backup solution.

As an example, if you have a web application, you will most probably want to create an image containing all the installed software you need on it (e.g., Apache, Rails), since you will not change that frequently. You could save the web content itself in a volume so you can update it and make backups of it regularly.

> EBS volumes are quite reliable, but you can't trust them to never die on you. An EBS volume is spread over several servers in different availability zones, comparable to RAID. Some people use a couple of volumes to create their own RAID, but according to AWS, "mirroring data across multiple Amazon EBS volumes in the same availability zone will not significantly improve your volume durability." However, taking a snapshot basically resets the volume, and a volume is more likely to fail when it is older.
>
> We use snapshots as a backup/restore mechanism; we take regular snapshots and hardly have any volume failures. As the EBS documentation states, "The durability of your volume depends both on the size of your volume and the percentage of the data that has changed since your last snapshot.[...] So, taking frequent snapshots of your volume is a convenient and cost effective way to increase the long term durability of your data."

When an instance is launched, AWS assigns it an IP address. Every time the instance is stopped, this IP address vanishes as well—not really ideal. AWS' solution is the use of Elastic IPs (EIPs). You can request an EIP and assign it to your instance every time you start/launch it again, so you always keep the same IP address. For now, you can assign only one EIP per instance. The interesting thing about EIPs is that they are free only when used, a nice incentive not to waste resources. In case you plan to send mail, an EIP also gives you the opportunity to ask AWS to lift email restrictions.

> An EIP is referred to by its IPv4 address, for example 184.72.235.156. In the Console, but also with the command-line tools, all you see is that address. But if you assign this EIP to an instance, you see that the instance's *public DNS* changes to `ec2-184-72-235-156.compute-1.amazonaws.com`. This address refers to the private IP address internally and the public IP externally. For data transfers between instances using the public IP, you will pay the regional data transfer rates. So if you consistently use the DNS name related to the EIP, you not only reduce the network latency, you also avoid paying for unnecessary data transfers.

www.it-ebooks.info

We can do most of this work from the Console, but all of it can be executed from the command line. This means we can script an instance to provision itself with the proper EBS volumes and associate the right EIP. If your instance dies, for whatever reason, this will save you valuable minutes figuring out which EIP goes where and which EBS volume should be attached to what device.

Let's look at how to create and use EBS volumes in your instance and assign an EIP to it.

Creating and using an EBS volume. Remember that we didn't specify the availability zone when launching our instance? We need to figure out which availability zone our instance ended up in before we create an EBS volume. Our instance ended up in `us-east-1b`. A volume can only be attached to one instance at the same time, and only if the volume and instance are in the same availability zone. An instance can have many volumes, though. Kulitzer will have a lot of user content, but we plan to use Amazon S3 for that. We'll show later how we use S3 as a content store for this application. For now, it is enough to know that we don't need a large volume: a minimum of 1 GB is enough.

> You can't enlarge a volume directly. If you need to make your volume bigger, you need to create a snapshot from it, then create a bigger volume from that snapshot. Then you will need to tell the filesystem that your partition is larger, and the way to do that depends on the specific filesystem you are using. For XFS like we are using in the examples, it's quite simple—you use the command `xfs_growfs /mount/point`.

Once your volume is available, you can attach it to the instance right from the same screen, by specifying your device. Because it is the first volume we attach, `/dev/sdf` is the most logical choice. Within moments you should see the device. We want an XFS volume, mounted at `/var/www`. On Ubuntu, this is all it takes (provided you installed the required packages and your mountpoint exists):

```
$ mkfs.xfs /dev/sdf
$ mount -t xfs -o defaults /dev/sdf /var/www
```

> If you are using Ubuntu, you will need to install the package for XFS with `apt-get install xfsprogs` and create the directory */var/www* (if it doesn't already exist). `mkfs.xfs` will build an XFS filesystem on the volume, which can then be mounted.
>
> If you are used to adding your mounts to `/etc/fstab`, it is better not to do that in Ubuntu Lucid 10.04 and later. In previous versions, the boot process continues even if it is unable to mount all specified mount points. Not anymore. If later versions of Ubuntu encounter an unmountable mountpoint, it just halts, making your instance unreachable. We just specify the full mount command in our startup script, which we'll show later.

www.it-ebooks.info

Creating and associating an EIP. There is really not much to it. You create an EIP from the AWS Console by clicking on Allocate New Address in the Elastic IPs section (Figure 2-8). Then you can associate it to a running instance by clicking on Associate. It is basically making sure you keep your IP addresses, for example, for DNS. Keep in mind that when you stop an instance, the EIP is disassociated from the instance.

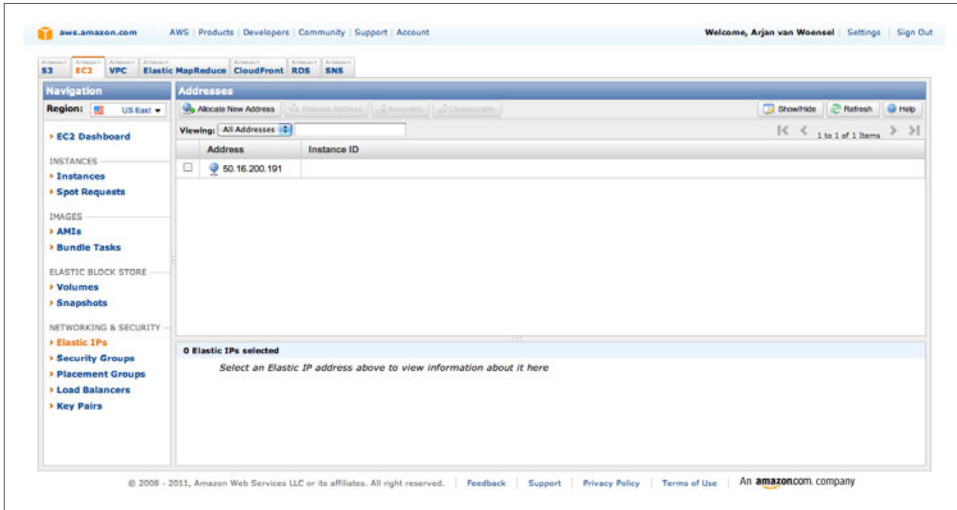


Figure 2-8. Elastic IPs in the AWS Console

Installing the software. Now would be a good time to install your web server and other software you need, such as Apache and Rails in our example.

Creating a custom image

Your instance is ready and it works. You have an Ubuntu server that you can connect to through SSH, HTTP, and HTTPS. This instance has an EBS volume attached to it, which will contain your web application content, and a static IP associated using an EIP. Your web server has been set up as well.

The only thing missing is to set up the database. But before doing that, it is better to prepare for other situations and save your work in a new image. Suppose you want to scale up, to use the much better performing High-CPU Medium instance; in this case, you will need an image to launch it from. Or suppose you lose your instance entirely and you don't want to recreate the instance from scratch. Apart from an unhappy customer, boss, or hordes of angry users, it is just tedious to do everything over again.

To scale or recover quickly, you can create a custom image. With this image, you can easily launch another instance with all software installed. Creating an image is straightforward. You can either do it from the Console or use the following commands:

```
# stop the instance explicitly, and detach the volume
$ ec2-stop-instances i-8eda73e4
```


www.it-ebooks.info

```
$ ec2-detach-volume vol-c00177a9

# create an image for this instance, with the given name and description
$ ec2-create-image i-8eda73e4 -n app-server-20100728 -d 'Rails Application Server'

# start the instance again, attach the volume and associate elastic ip
$ ec2-start-instances i-8eda73e4
$ ec2-attach-volume vol-c00177a9 -i i-8eda73e4 -d /dev/sdf
$ ec2-associate-address 184.72.235.156 -i i-8eda73e4
```

> This might be the first time you use the command-line tools. If you do not see what you expect to see, like your instance or volumes, you might be working in a region other than the default region. The command-line tools accept `--region` to work somewhere else. For example, to list instances in Europe (Ireland), you can use `ec2-describe-instances --region eu-west-1`. For a full list of available regions, you can use the command we used to test the command-line tools: `ec2-describe-regions`.

There are a couple of things you need to know. First, *if you don't detach the volume,* `ec2-create-image` *will create a snapshot of the volume as well*. When launching a new instance it will not only create a new root volume, but also a new volume with your application. For this setup, you don't want that; you will use the existing volume. Second, stopping the instance is not really necessary, according to AWS. You can even specify `--no-reboot` when creating the image, but the integrity of the filesystem cannot be guaranteed when doing this. We will take no chances: we'll stop the instance explicitly. And finally, we don't disassociate the EIP, as this is done automatically.

We could launch a new instance, and perhaps it is even a good idea to test whether the image works. But we have to do one other thing. We don't want to attach volumes manually every time we launch an instance of this image. Also, attaching the volume does not automatically mount the volume. Furthermore, we don't want to associate the EIP ourselves—we want to launch an instance that provisions itself.

Provisioning the instance at boot/launch. For an instance to provision itself, we need some way to execute the proper commands at boot/launch time. In Ubuntu, we do this with init scripts. For other operating systems, this might be slightly different, but the general idea can be applied just the same. On your instance, create the file */etc/init.d/ec2*, making sure it is executable and contains the following script:

```
#!/bin/bash
### BEGIN INIT INFO
# Provides:          ec2-instance-provisioning
# Required-Start:    $network $local_fs
# Required-Stop:     $apache2
# Should-Start:      $named
# Should-Stop:
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
```


www.it-ebooks.info

```
# Short-Description: ec2 housekeeping
# Description:       attach/detach/mount volumes, etc.
### END INIT INFO

#
# ec2-elastic - do some ec2 housekeeping
#     (attaching/detaching volumes, mounting volumes, etc.)
#

export JAVA_HOME='/usr'
export EC2_KEY_DIR=/root/.ec2
export EC2_PRIVATE_KEY=${EC2_KEY_DIR}/pk-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_CERT=${EC2_KEY_DIR}/cert-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_HOME='/root/ec2-api-tools-1.3-53907'
export EC2_URL="https://eu-west-1.ec2.amazonaws.com"
PATH=$PATH:$HOME/bin:$EC2_HOME/bin
MAX_TRIES=60

prog=$(basename $0)
logger="logger -t $prog"
curl="curl --retry 3 --silent --show-error --fail"
# this URL gives us information about the current instance
instance_data_url=http://169.254.169.254/latest
region="eu-west-1"
elastic_ip=184.72.235.156

vol="vol-c00177a9"
dev="/dev/sdf"
mnt="/var/www"

# Wait until networking is up on the EC2 instance.
perl -MIO::Socket::INET -e '
 until(new IO::Socket::INET("169.254.169.254:80"))
 {print"Waiting for network...\n";sleep 1}
' | $logger

# start/stop functions for OS

start() {
    ctr=0
    # because the instance might change we have to get the id dynamically
    instance_id=$($curl $instance_data_url/meta-data/instance-id)

    /bin/echo "Associating Elastic IP."
    ec2-associate-address $elastic_ip -i $instance_id --region=$region

    /bin/echo "Attaching Elastic Block Store Volumes."
    ec2-attach-volume $vol -i $instance_id -d $dev --region=$region

    /bin/echo "Testing If Volumes are Attached."
    while [ ! -e "$dev" ] ; do
        /bin/sleep 1
        ctr=`expr $ctr + 1`
        # retry for maximum one minute...
        if [ $ctr -eq $MAX_TRIES ]; then
```


www.it-ebooks.info

```
            if [ ! -e "$dev" ]; then
                /bin/echo "WARNING: Cannot attach volume $vol to $dev --
                    Giving up after $MAX_TRIES attempts"
            fi
        fi
    done

    if [ -e "$dev" ]; then
        if [ ! -d $mnt ]; then
            mkdir $mnt
        fi

        /bin/echo "Mounting Elastic Block Store Volumes."
        /bin/mount -t xfs -o defaults $dev $mnt
    fi
}

stop() {
    /bin/echo "Disassociating Elastic IP."
    ec2-disassociate-address $elastic_ip --region=$region

    /bin/echo "Unmounting Elastic Block Store Volumes."
    /bin/umount $mnt

    ec2-detach-volume $vol --region=$region

}


case "$1" in
    start)
        start
        ;;

    stop)
        stop
        ;;
    restart)
        stop
        sleep 5
        start
        ;;
    *)
        echo "Usage: $SELF {start|stop|restart}"
        exit 1
        ;;

esac

exit 0
```

You will need to have the EC2 command-line tools installed on your instance. Replace the environment variables accordingly. Remember to change to the region you are using if it's not `us-east-1`, and use your EIP and volume ID.

www.it-ebooks.info

Make sure the operating system actually executes the script at startup. In Ubuntu, you can do it like this:

```
$ update-rc.d ec2 defaults
```

Create a new image the same way as before and try it by launching an instance (make sure you choose the right availability zone). You should have a new instance with a brand new instance identifier, but the same volume for the application and EIP address (if you stop now, you don't have to detach the volume anymore). If you want to scale up or if you run into trouble, just stop or terminate your old instance and start a new one.

> One last tip before we continue with the database. When going through the Request Instances Wizard, you may have wondered what a spot instance is. AWS describes them as follows:
>
> > They allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current Spot Price. The Spot Price changes periodically based on supply and demand, and customers whose bids meet or exceed it gain access to the available Spot Instances.
>
> You can configure your spot request to be `closed` when the spot price goes above your maximum price, or to keep the request open and create another instance the next time the price goes below your maximum.

User data. Another often-used way to make your instance do some particular work at launch time is called *user data*. When you launch an EC2 instance, you give it some data in the form of an executable shell script to be performed at boot. You can use this to do some additional configuration or even installation of software.

You can create this script on your development machine and pass it in the command that will launch your instance:

```
$ ec-run-instances --user-data-file my-local-script.sh ami-714ba518
```

The drawback of this method of provisioning your instance is that you cannot reprovision a running instance. If, for example, you upgrade your application and you need to update your instance with the new sources, you will have to add the script to the instance (or image) explicitly, making it pointless to send it through user data. If you don't have the need to reprovision or if launching a new instance for upgrading is OK for your system, this method is fine.

RDS Database

If you've gotten this far, you have probably noticed our love for Amazon RDS. In case you didn't, let us reiterate: we love this service. Setting up and maintaining a MySQL database doesn't appear to be too difficult. But setting it up properly, with a backup/

www.it-ebooks.info

restore mechanism in place, perhaps even replicated for higher availability and tuned for optimal performance, is difficult. As your traffic grows, it is inevitable that you will have to scale up. And as your application becomes more important, you will want to implement replication to minimize downtime, and you will have to keep the database software up to date.

Amazon introduced RDS not too long ago. RDS provides almost everything you need to run a production-grade database server without the immediate need for a database administrator (DBA). Often, the DBA also helps with optimizing the schemas. Of course, RDS is not capable of doing that for you, but it will take care of backups, so you will not lose more than five minutes of data in case of a crash, and you can go back in time to any second during a period of up to the last eight days. It will automatically upgrade the MySQL database software for you, provide enhanced availability in multiple zones, and read replicas to help you scale.

> We need to get a bit technical here, as there is a very important, not-too-well-documented feature: if you use MyISAM as your storage engine, you do *not* get an important part of the backup functionality.
>
> There are two kinds of backups: snapshots and Restore to Point in Time. The first is manual and tedious; the second is the best thing since sliced bread. Backups are available with MyISAM, but you have to make your databases read-only before you take a snapshot. If you want to use Restore to Point in Time backups, make sure you use InnoDB as the storage engine for *all* the tables in *all* the databases in your RDS instances.
>
> When importing your old database, you can start with InnoDB by not specifying the storage engine. If you want to migrate from MyISAM to InnoDB, take a look at the MySQL documentation.

Before we continue, make sure you sign up for Amazon RDS. At this time, almost all of the functionality of RDS is available through the Console. What's missing is important, however, and we need those features to set up the DB instance the way we want. You already downloaded the Amazon RDS Command Line Toolkit and added the necessary environment variables to your script in the previous section, so nothing is stopping you from creating an RDS instance.

Creating an RDS Instance (Launching the DB Instance Wizard)

This wizard is familiar—it looks a lot like the one for launching EC2 instances. This wizard, though, does not allow you to create some of the necessary assets on the fly. But in contrast to EC2, you can change all of the options for a running DB instance. Changing some of the options requires a restart of the database instance, either immediately after making the change or during the maintenance window you will choose later.

www.it-ebooks.info

It is a good idea to create a DB security group first, if you don't want to use the default. You can create the DB security group through the AWS Console and add the authorizations. An authorization is a security group from an EC2 account that allows you to share RDS instances easily across multiple accounts. Alternatively, you can specify a CIDR/IP, much like we did for the EC2 security groups. For now, we will allow everyone access to the DB instances in this group. Later, we'll restrict that access again. You can add a DB instance to multiple DB security groups, giving you the necessary flexibility in case you work with multiple EC2 accounts.

The DB Instance Details screen allows you to set the most important options. *Multi-Availability Zone* (multi-AZ) deployment is the high-availability option of RDS, creating a second, replicated DB instance in another zone. It is twice as expensive, but it gives you automatic failover in case of emergency and during maintenance windows.

You also have to indicate the allocated storage. The storage you allocate does not restrict you in what you actually consume; it only means that if you exceed the amount of allocated storage, you pay a higher fee.

The Additional Configuration page of the wizard allows you to specify a database name if you want an initial database to be created when the instance is launched. Otherwise, you can create your database(s) later. You can choose a port other than the default 3306. You can choose an availability zone, but that doesn't matter very much because *network latency between zones is comparable to that within a zone*. Later, we'll create a DB parameter group; for now, we'll use the default. Finally, you must indicate which DB security groups you want to add this instance to.

The Management Options screen gives you reasonable defaults. If you want to be able to restore a database from a week ago, for example, you can override the default one day in the Backup Retention Period. Longer than a week is not possible, but you can create DB snapshots from which you can easily launch DB instances. RDS makes backups every day, whereby the log file is flushed and data is stored. During the backup window, the database is read-only, blocking write operations until the backup is completed. It doesn't take more than a couple of minutes. If you set the Backup Retention Period to 0, you disable backups altogether. The maintenance window is scheduled sometime during the week. AWS reserves the right to automatic updates of the underlying MySQL. It is not possible to disable this feature.

If you don't want backups and maintenance to interfere with your operation, you can choose to run your DB instance in multi-AZ mode. Some of the advantages of this are that backups are done on the replicated DB instance and that maintenance is conducted on the replica, after which the RDS automatically performs a failover to do the maintenance on the other instance.

You are now ready to launch (Figure 2-9).

www.it-ebooks.info

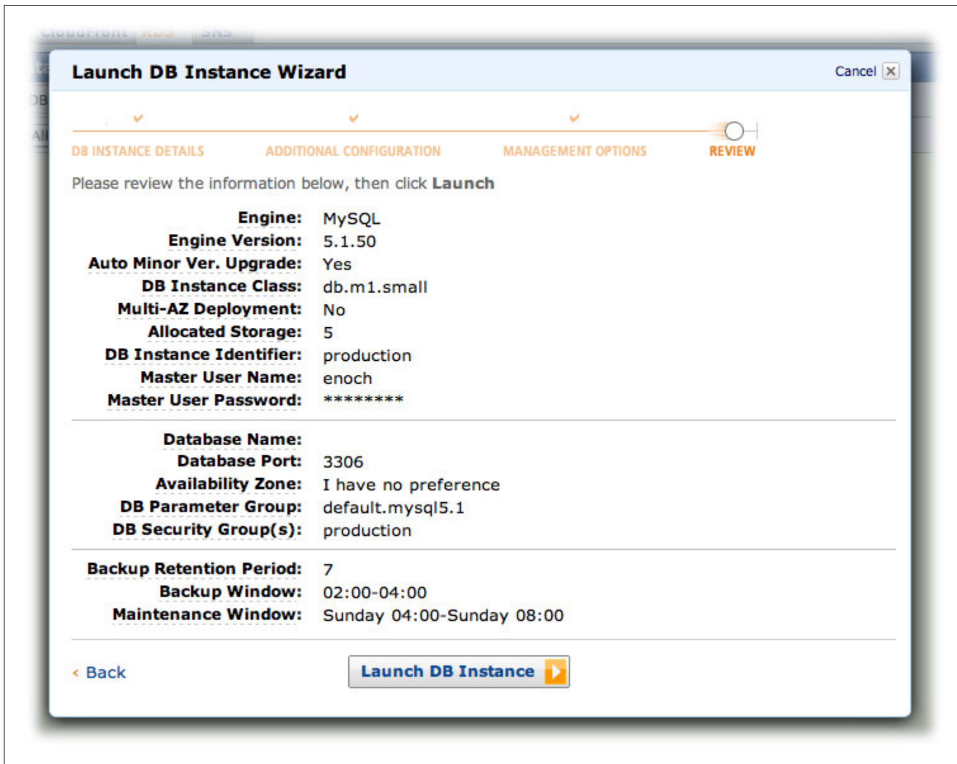


Figure 2-9. Launch from the DB Instance Wizard

Of course, as always, you can do the same on the command line:

```
$ rds-create-db-security-group production \
        --db-security-group-description \
            'this RDS is only available on the necessary ports'
$ rds-authorize-db-security-group-ingress production \
        --cidr-ip 0.0.0.0/0
$ rds-authorize-db-security-group-ingress production \
        --ec2-security-group-name web \
        --ec2-security-group-owner-id 457964863276

$ rds-create-db-instance production \
    --engine MySQL5.1 \
    --db-instance-class db.m1.small \
    --allocated-storage 5 \
    --master-username kulitzer \
    --master-user-password sarasa1234 \
    --db-security-groups production \
    --backup-retention-period 3
```


www.it-ebooks.info

Notice that in this example we are giving access to the world by using the CIDR 0.0.0.0/0 for convenience while setting it up, but we will have to remove it later. The equivalent in the AWS Console looks like Figure 2-10.

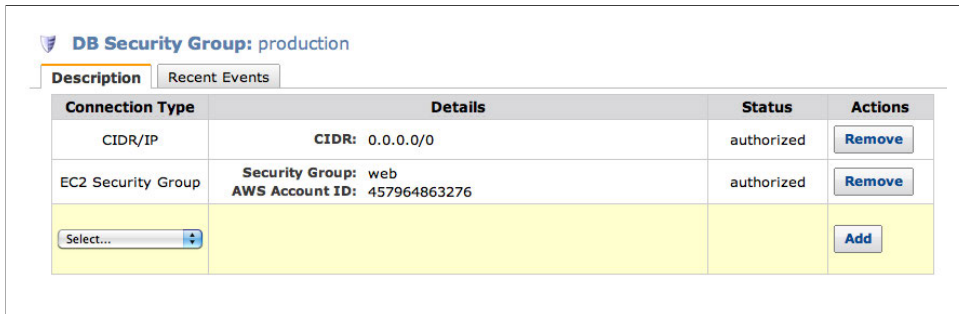


Figure 2-10. Configure a DB security group

> The DB instance classes resemble EC2 instance types. One of the missing classes is Medium. For EC2, the Medium instance is superior to the Small instance, and if you have the money, always go with Medium over Small. But with RDS we got lucky—considering the responsiveness and performance of a Small DB instance, it appears as if it is running on a High-CPU Medium EC2 instance, or something very similar. And all this for a price that is only slightly higher than what you pay for a Small EC2 instance. If you need a relational database and MySQL is an option, you need a seriously good reason not to do this. (Sorry, did that just sound too much like a fanboy?)

Is This All?

If it's this simple, how can it be so good? There's not much else to it, actually. You can create snapshots of your DB instance. You don't need that for backup/restore purposes if you use the backup facility of RDS. But you can create snapshots that you can use to create preconfigured database instances with databases and data. Figure 2-11 shows the Console screen for creating a DB instance from a snapshot. You can also track what you or AWS is doing to your database with DB Events.

This is not all. Although the basic configuration of RDS instances is sufficient for many cases, it is often required to change engine parameters. As you don't have *super* privileges, you need another way to change engine parameters. RDS offers *DB parameter groups* to change some—but certainly not all—of the available parameters. A DB parameter group contains engine configuration values that you can apply to one or more DB instances. AWS discourages users from using DB parameter groups, but it is necessary for some basics like slow query logging.

www.it-ebooks.info

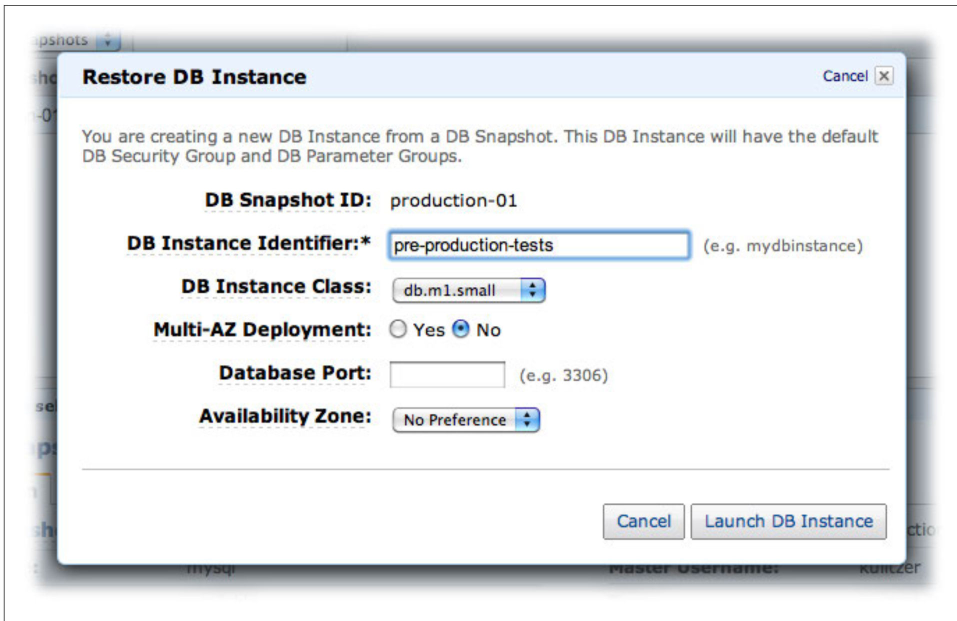


Figure 2-11. Create a DB instance from a snapshot

At this moment, you can create the DB parameter group from the Console, but you cannot modify the parameter values there. If you want to change actual parameters, you can do it using the command-line tools (or API). Enabling the slow query log, for example, is done like this:

```
$ rds-create-db-parameter-group production  \
        --description='custom parameter settings, for example slow_query_log' \
        --engine=MySQL5.1
$ rds-modify-db-parameter-group production \
        --parameters="name=slow_query_log, value=1, method=immediate"
$ rds-modify-db-parameter-group production \
        --parameters="name=long_query_time, value=1, method=immediate"
$ rds-modify-db-instance production \
        --db-parameter-group-name=production \
        --apply-immediately
$ rds-reboot-db-instance production
```

Notice that in this case, a reboot is needed because we are assigning a new DB parameter group to the instance. If you specify the parameter changes with `method=immediate`, they will be applied immediately to all database instances in that parameter group, only for the parameters of type `dynamic`. If you use `method=pending-reboot` or for parameters of type `static`, changes will be applied upon next reboot.

www.it-ebooks.info

During our work with RDS, we once needed a MySQL database server for which RDS was not sufficient. The JIRA issue tracking system requires MySQL's default storage engine to be InnoDB because it uses the READ-COMMITTED transaction level. The problem we encountered had to do with the combination of binary logging (which RDS uses for backups/replication) and InnoDB. MySQL only supported a binary logging format of type ROW, and we couldn't change this particular parameter.

But the version of MySQL our RDS instance was running was 5.1.45. This particular combination of features is supported in version 5.1.47 and later. It was also interesting to see that Ubuntu's default MySQL package had version 5.1.41. We did not want to wait, because we didn't know how long it would take. We set up a simple MySQL database on the instance itself with a binary log format of ROW. At the time of this writing, RDS supports engine version 5.1.50.

S3/CloudFront

Most simple web applications are built as three-tier architectures. You might not even be aware of this, as it is (most of the time) unintentional. The three tiers of these applications are:

1. Data (usually kept in a relational database)
2. Logic (dynamic content generated in a web or application server)
3. Presentation (static content provided by a web server)

Over time, web development frameworks slowly lost the distinction between the logic and presentation levels. Frameworks like PHP, Ruby on Rails, and Django rely on Apache modules, effectively merging the logic and presentation levels. Only when performance is an issue will these two layers be untangled, mainly because the overhead of Apache is not necessary to serve semistatic content.

But there is another alternative: CloudFront. Amazon CloudFront is a content distribution network, designed to bring static content as close to the end user as possible. It has *edge locations* all over the world, storing your files and delivering them upon request. Perhaps we don't need the edge locations yet, but we can use CloudFront for its scalability *and* to offload our application server. We'll use CloudFront as the presentation tier in our three-tiered architecture.

Setting Up S3 and CloudFront

Amazon S3 is a regional service, just like EC2 and RDS, while CloudFront is a global service. S3 works with *buckets*, and CloudFront works with *distributions*. One CloudFront distribution exposes the content of one S3 bucket (one bucket can be exposed

www.it-ebooks.info

by multiple distributions). CloudFront can serve content for download and streaming, provided the content allows for streaming.

At this moment, we want to store and serve the static content from S3/CloudFront. Because our application server is in the US East region (the default), we create our S3 bucket in the same region, which in the Console is called US Standard (Figure 2-12).

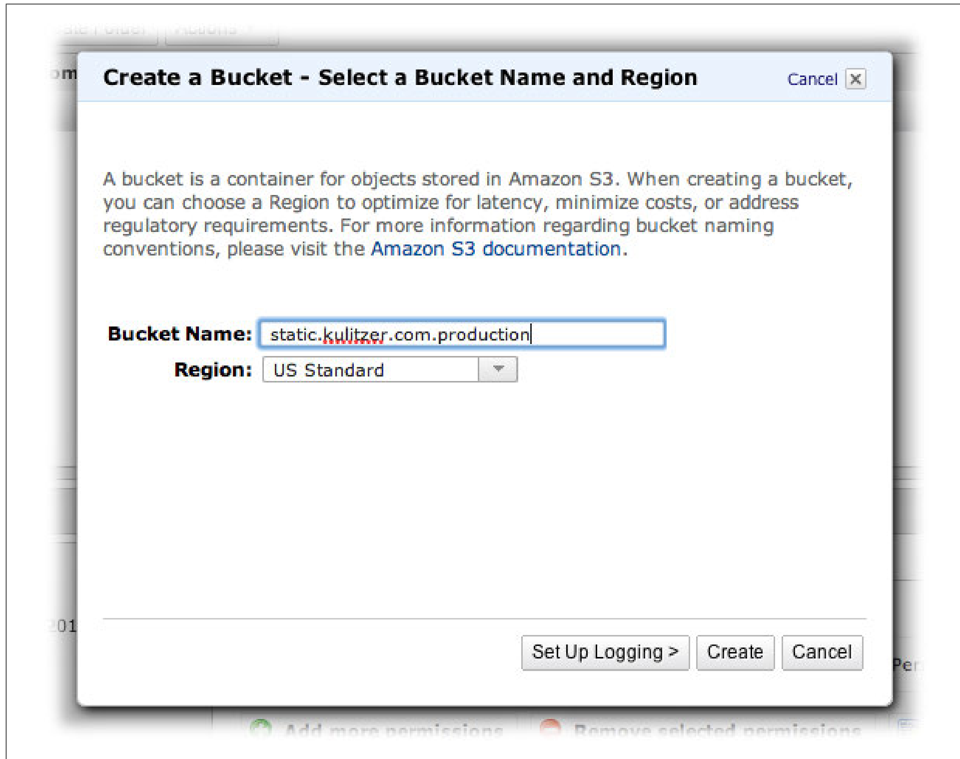


Figure 2-12. Create an S3 bucket

> S3 was only recently added to the Amazon AWS Console. We have worked with quite a few S3 applications, but are slowly switching to using the Console for our S3-related work. It is sufficient because we don't need it very often and it is close to CloudFront, so we can quickly switch between the two.

With our S3 bucket, we can create the CloudFront distribution (Figure 2-13). One of the interesting things about CloudFront is that you can easily expose one distribution through multiple domains. We plan to have all static content accessible through `static[0..3].kulitzer.com`. We will configure Rails to use these four domains to speed up page load times, since the browser will then be able to download several assets in

www.it-ebooks.info

parallel. This technique is one of the core features in Rails. For other frameworks, you might have to do some extra work to use several different domains.

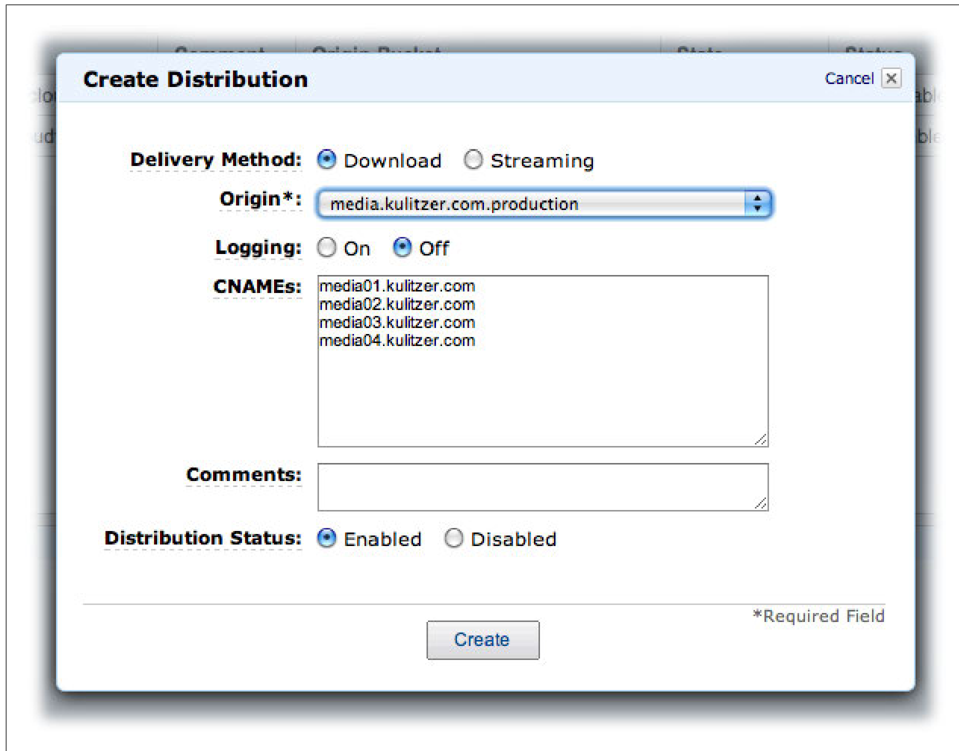


Figure 2-13. Create a CloudFront distribution

It might take a little while before your distribution is enabled. AWS has already determined the domain for the distribution: `d2l9lx40wmdm9x.cloudfront.net`. Using this domain, we can add the CNAME records to our DNS.

Static Content to S3/CloudFront

The distribution we just created serves content from the S3 bucket. We also created four domains pointing to this bucket. *If we add content to the S3 bucket, it will be available from our CloudFront distribution through these domains*. We want to serve our JavaScript, CSS, and other static content like images from CloudFront. In Rails, this is fairly simple if you use the URL-generating helpers in AssetHelper. Basically, the only two things you need to do are to configure `config.action_controller.asset_host` to point to the proper asset hosts, and upload the files to S3. We set the configuration in `config/environments/production.rb` like this:

```
# Enable serving of images, stylesheets, and javascripts from an asset server
config.action_controller.asset_host = "http://static%d.kulitzer.com"
```


www.it-ebooks.info

Rails will replace the `%d` with 0, 1, 2, or 3.

After uploading the static content to the S3 bucket (make sure you make these objects public), the Kulitzer logo is served from *http://static2.kulitzer.com/images/logo.jpg* or one of the other domains. The result is that the assets a page needs are evenly requested from the different domains, allowing your browser to download them in parallel. Figure 2-14 shows our Kulitzer site set up using EC2, RDS, S3, and CloudFront.

> Apart from increasing the scalability of your infrastructure, you also implemented one of the patterns for optimizing the performance of your web app. For Kulitzer, we use AssetPackager to optimize and simplify working with CloudFront even more. AssetPackager merges the JavaScript and CSS files into one, speeding up load times.

One important aspect of CloudFront is how it distributes the assets from S3 to the edge locations. The edge locations get new copies within 24 hours at the most, and usually much quicker than that. You can override this behavior by specifying `Cache-Control`, `Pragma`, or `Expires` headers on the object in S3. If you specify an expiration time of less than one hour, CloudFront uses one hour. If you want to force a particular object to be changed immediately, you can *invalidate* it by calling the Invalidation API. Invalidating a file removes it from all the CloudFront edge locations. The documentation says it is supposed to be used under exceptional circumstances, such as when you find an encoding error in a video and you need to replace it.

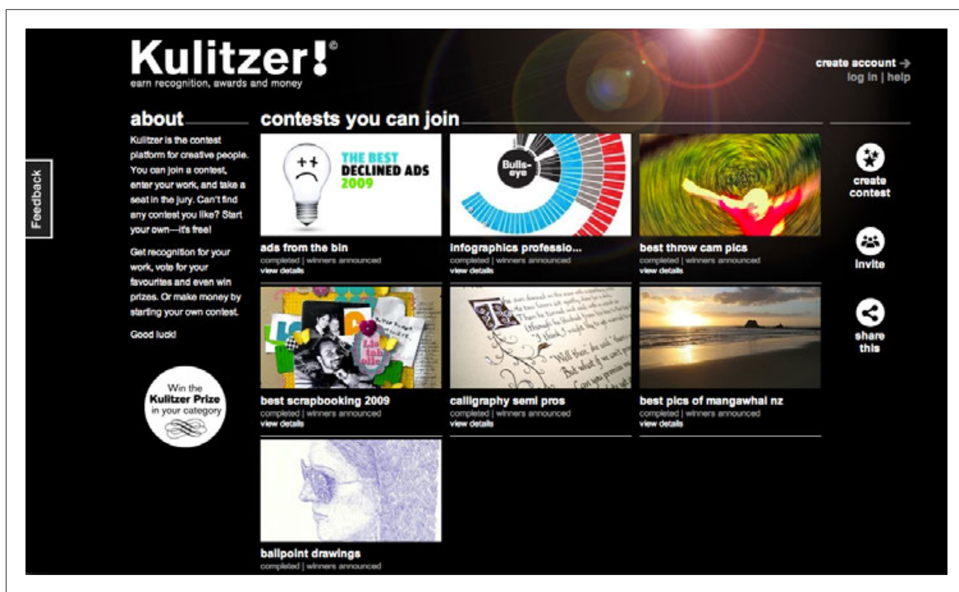


Figure 2-14. Kulitzer.com

www.it-ebooks.info

Case study: Publitas—CloudFront to the rescue

Publitas has developed a web application called ePublisher that enables its customers to publish rich content online, starting from a PDF. Right after the summer of 2010, Publitas found its dream customer. This customer was experiencing heavy web traffic and needed help to keep up.

With a large database of interested customers, Publitas figured it would be good to start relatively slowly, and it sent out email messages to 350,000 people. The message pointed people to the online brochure consisting of descriptions of products with rich media content like audio and video.

The response was overwhelming, and the servers couldn't handle this. Luckily, ePublisher has an export feature, and the entire brochure was quickly exported to S3 and exposed through CloudFront. Everything was on a subdomain, so the system was only waiting for DNS to propagate the changes while rewriting incoming requests. And everything worked flawlessly. Publitas was happy and the customer was happy.

This particular brochure saw nearly 440,000 unique IPs and 30 terabytes of traffic in the first month.

Making Backups of Volumes

Backups are only there for when everything else fails and you need to be able to go back in time. Backing up everything is often problematic, because it requires several times more storage than you need to just run the application.

With snapshots, we have the means to easily and very quickly take a snapshot of a volume. You can later restore this snapshot to another volume. Snapshots are incremental, not at the file level, but at the block level. This means you don't need 100 times as much storage for 100 backups—you probably need just a couple of times the size of your volume. However, this is difficult to verify, because we have never found out where to see how much storage we use for our snapshots.

To finish our backup/restore mechanism, we need a way to clean up the old snapshots. If we can take snapshots for which we can set the expiration date to one week or one month in the future, we have enough. We created a couple of scripts that use SimpleDB for snapshot administration, and the EC2 command-line utilities to take and delete snapshots.

For our backup mechanism, we use SimpleDB to administer expiration dates. And we want our solution to be no more than two scripts, one for taking a snapshot and one for expiring snapshots. In Linux, there are some powerful tools like *date* that we use to calculate dates. Use the command `man date` if you want to know about calculating dates. Furthermore, we need the EC2 command-line tools and a Perl command-line tool for SimpleDB. (See the beginning of this chapter for instructions on installing EC2.)

www.it-ebooks.info

Installing the Tools

Let's take some time to go over the installation of the SimpleDB client. We haven't introduced SimpleDB yet, and it uses some tools that are not self explanatory. So, here we go:

1. First, download (but don't install) the Perl Library for Amazon SimpleDB.
2. Go to *http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1136*.
3. Look at the prerequisites and install them:

```
$ sudo perl -MCPAN -e 'install Digest::SHA'
$ sudo perl -MCPAN -e 'install XML::Simple'
$ sudo perl -MCPAN -e 'install Bundle::LWP'
$ sudo perl -MCPAN -e 'install Crypt::SSLeay'
```

4. Go to the SimpleDB Cli page.
5. Look for the INSTALLATION section and install the following prerequisites:

```
$ sudo perl -MCPAN -e 'install Getopt::Long'
$ sudo perl -MCPAN -e 'install Pod::Usage'
$ sudo perl -MCPAN -e 'install Digest::SHA1'
$ sudo perl -MCPAN -e 'install Digest::HMAC'
$ sudo perl -MCPAN -e 'install XML::Simple'
```

6. Install the Amazon SimpleDB Perl library following the installation guide.
7. Install the Amazon SimpleDB Perl library and the SimpleDB command-line interface using the following:

```
$ unzip AmazonSimpleDB-2009-04-15-perl-library.zip
$ sitelib=$(perl -MConfig -le 'print $Config{sitelib}')
$ sudo scp -r Amazon-SimpleDB-*-perl-library/src/Amazon $sitelib

$ sudo curl -Lo /usr/local/bin/simpledb http://simpledb-cli.notlong.com
$ sudo chmod +x /usr/local/bin/simpledb
```

Before you can continue, you need to create a domain (there is one irritating deficiency in the SimpleDB command-line interface, which is that it does not accept a region and always takes the default `us-east-1` region):

```
$ export AWS_ACCESS_KEY_ID='your access key id'
$ export AWS_SECRET_ACCESS_KEY='your secret access key'
$ simpledb create-domain snapshot
```

Running the Script

The backup script is called with one parameter to indicate the expiration in a human-readable format, for example "24 hours." We will execute these backups from the instance itself. We use simple cron jobs, which we'll show later, to create an elaborate backup scheme. This is the entire backup script:

www.it-ebooks.info

```
#!/bin/bash
#
# install http://code.google.com/p/amazon-simpledb-cli/
# and http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1136
# WARNING: make sure to install the required packages of the second as well

# specify location of X.509 certificates for the ec2 command line tools
export EC2_KEY_DIR=/root/.ec2
export EC2_PRIVATE_KEY=${EC2_KEY_DIR}/pk-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_CERT=${EC2_KEY_DIR}/cert-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_ACCESS_KEY='AKIAIGKECZXA7AEIJLMQ'
export AWS_ACCESS_KEY_ID='AKIAIGKECZXA7AEIJLMQ'
export EC2_SECRET_KEY='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'
export AWS_SECRET_ACCESS_KEY='w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn'
export EC2_USER_ID='457964863276'
export EC2_HOME='/root/ec2-api-tools-1.3-53907'
export JAVA_HOME='/usr'
PATH=$PATH:$HOME/bin:$EC2_HOME/bin:/usr/local/bin

region="us-east-1"

# if called with a parameter that is accepted by 'date --date'
# it creates a date based on that value. if it is empty we take
# a default expiration of 24 hours
offset=$1
if [ "${offset}" == "" ]
then
    offset="24 hours"
fi

expiration=$(date -u --date="${offset}" +"%Y-%m-%d %H:%M:%S")
if [ "$expiration" == "" ]
then
    exit 0
fi

vols=( "vol-c00177a9" )

mountpoints=( "/var/www" )

for ((i = 0; i < ${#vols[@]}; i++))
do
    xfs_freeze -f ${mountpoints[i]}
    snapshot=($(ec2-create-snapshot ${vols[i]} --region $region))
    xfs_freeze -u ${mountpoints[i]}

    # now add an item to the SimpleDB domain
    # containing the snapshot id and its expiration
    /usr/local/bin/simpledb put snapshot ${snapshot[1]} expires="${expiration}"
done
```

Notice that we make sure the mountpoints are read-only when taking the snapshot. This is especially for databases, as they might come to a grinding halt when their binary files and logfiles are inconsistent. The `vols` and `mountpoints` variables are arrays. You can give any number of volumes, as long as the corresponding mountpoints are given.

www.it-ebooks.info

The script will continue regardless, but snapshots are taken without the mountpoint frozen in time. You will need to create a domain in SimpleDB called *snapshot*, where we add an item.

To illustrate how easy it is to create our backup scheme, here is the cron, which schedules a process to delete expired backups daily and to make backups every three hours, daily, weekly, and monthly:

```
# m h  dom mon dow   command
@daily      /root/ec2-elastic-backups/ec2-elastic-expire > /dev/null 2>&1

0 */3 * * * /root/ec2-elastic-backups/ec2-elastic-backup "24 hours" > /dev/null 2>&1
@daily      /root/ec2-elastic-backups/ec2-elastic-backup "7 days" > /dev/null 2>&1
@weekly     /root/ec2-elastic-backups/ec2-elastic-backup "1 month" > /dev/null 2>&1
@monthly    /root/ec2-elastic-backups/ec2-elastic-backup "1 year" > /dev/null 2>&1
```

And here is the script that deletes the expired snapshots:

```
#!/bin/bash
#
# install http://code.google.com/p/amazon-simpledb-cli/
# and http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1136
# WARNING: make sure to install the required packages of the second as well

export EC2_KEY_DIR=/root/.ec2
export EC2_PRIVATE_KEY=${EC2_KEY_DIR}/pk-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_CERT=${EC2_KEY_DIR}/cert-4P54TBID4E42U5ZMMCIZWBVYXXN6U6J3.pem
export EC2_ACCESS_KEY='AKIAIGKECZXA7AEIJLMQ'
export AWS_ACCESS_KEY_ID='AKIAIGKECZXA7AEIJLMQ'
export EC2_SECRET_KEY='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'
export AWS_SECRET_ACCESS_KEY='w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn'
export EC2_USER_ID='457964863276'
export EC2_HOME='/root/ec2-api-tools-1.3-53907'
export JAVA_HOME='/usr'
PATH=$PATH:$HOME/bin:$EC2_HOME/bin:/usr/local/bin

region="us-east-1"

now=$(date +"%Y-%m-%d %H:%M:%S")

snapshots=$(simpledb select "select * from snapshot where expires < '${now}'")

for snapshot in $snapshots
do
    snap=`expr match "$snapshot" '.*\(snap-........\).*'`
    if [ -n "$snap" ]; then
        # remove the item from SimpleDB
        simpledb delete snapshot $snap
        # delete the snapshot itself
        ec2-delete-snapshot $snap --region $region
    fi
done
```

This is all it takes to create a backup strategy that creates point-in-time snapshots at three-hour intervals, keeping all of them for at least 24 hours, and some up to a year.

www.it-ebooks.info

Taking a snapshot of a reasonably sized volume takes seconds. Compare that to `rsync`-based backups; even when run incrementally, `rsync` can take quite some time to complete. Restoration of individual files is a bit more problematic; `rsync` must first create a volume from the snapshot and then look for the specific file. But it is a fail-safe, not fool-safe, measure.

> At this point, it is probably a good idea to stop all the EC2 and RDS instances you were using for practice, to keep your credit card safe.

In Short

Well, in this chapter, we did quite a bit of work!

We set up an AWS account with EC2, installed the necessary command-line tools, and started using the AWS Console. We introduced the concept of regions and availability zones. And we finally got our hands dirty launching an instance based on an Ubuntu image. For that, we created a key pair, which grants you access to the instance so you can start using it. We introduced the concept of security groups to specify who is allowed to connect to an instance and with which protocols.

The next thing we needed was a volume for the web application content, resembling a disk. We attached one to our instance. To assign a static IP address to our instance, we used an EIP. After setting all this up, we created a custom image with all our changes, including scripts for associating the EIP, attaching the volume and mounting it at boot, and cleaning up when shutting down the instance.

The next big thing we looked at was RDS, the MySQL database service of AWS. We discussed the advantages of RDS, including backups, automatic software upgrades, almost immediate scaling, and high availability. We launched a DB instance and set up the allowed machines with the DB security groups.

To allow global, fast access to static content, we used CloudFront, the content distribution network of AWS. By uploading our assets in an S3 bucket and pointing CloudFront to it, we made our files accessible all over the world.

Finally, we looked at an easy way to create backups of our volumes.

We now basically have an instance running with a web server and application on it, using a database and distributing static content efficiently all over the world. And we do proper backups of everything, including volumes.

If your application becomes more popular, with more users and load, you can take advantage of many of AWS' capabilities. That's what we'll start looking at in the next chapters.

www.it-ebooks.info

CHAPTER 3
Growing with S3, ELB, Auto Scaling, and RDS

We have done quite a lot in just a couple of chapters. We have explored what it means to design, build, and operate virtual infrastructures on AWS. We have looked at the opportunities it provides, and we have moved a real-world application, Kulitzer.com, to AWS. Although we have done many things that are usually very difficult on physical infrastructures, we have not yet looked at the biggest benefit of AWS: an *elastic* infrastructure that scales with demand.

With growing traffic, our initial setup will soon be insufficient. We know we can *scale up* with bigger machines, but we prefer to *scale out*. Scaling up is OK in certain situations. If you have a fixed upper limit in traffic—for example, if your users are internal to an organization—you can safely assume it is not necessary to go through the trouble of implementing load balancing and/or autoscaling. But if your application is public, and perhaps global, this assumption can be costly. Scaling up is also manual on AWS, requiring reboots and temporary unavailability of your instance. The same goes for the opposite, scaling down (Figure 3-1).

Scaling out, on the other hand, requires changes to your application. You have to make sure you can have several instances doing the same work, preferably on their own. If the instances can operate in their own context, it doesn't matter if you have three or seven instances. If this is possible, we can handle traffic with just the right number of instances. If the traffic increases, we launch some more instances; if the traffic decreases, we terminate some of the instances.

We don't have to worry about scaling out the presentation layer (recall the three-tier architecture introduced in the previous chapter). We use S3/CloudFront, which does all the scaling out and scaling in for us. And even if you chose to implement your own infrastructure for this, it is not that hard. Making the logic layer elastic is a bit harder, but not impossible. You basically have to *make sure all of the data the application server works on that needs to be shared by other application servers does not reside on the application server itself*. Think of sessions, for example, or user content. But the data

www.it-ebooks.info

layer is more problematic. For traditional Relational Database Management Systems (RDBMS), you will probably require a strategy of scaling up *and* scaling out. But introducing elasticity is almost impossible in this situation.

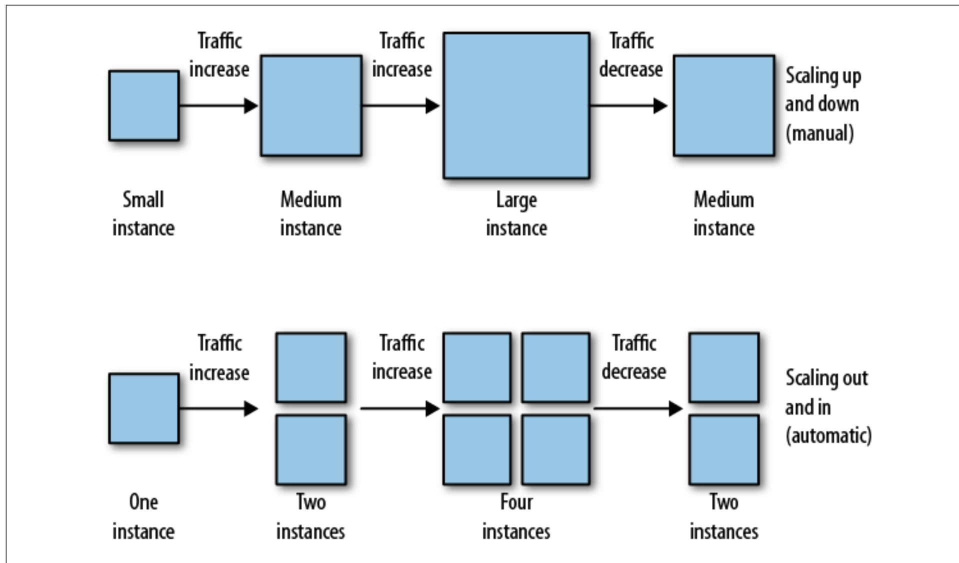


Figure 3-1. Scaling up versus scaling out

Preparing to Scale

Scaling an application on AWS is essentially the same as scaling any other infrastructure. As said before, if you use S3/CloudFront as your presentation tier, you don't have to worry about that. For the logic tier (Apache, in our case), you have to move out the data/information to shared storage. If you chose to implement your own presentation layer with thin HTTP servers like Nginx or Lighttpd, you can follow the same principle.

You have several options to move *filesystem-style data* to a shared storage solution (Table 3-1). Remember that an EBS volume can only be attached to one instance at a time, so it is not an option. The obvious choice if you don't have a Storage Area Network (SAN)/Network Attached Storage (NAS) at your disposal used to be to implement something like Network File System (NFS) or Server Message Block (SMB), but they are not very scalable. An alternative is to move this data to a database, relational or otherwise. In both cases, you don't really solve the scalability problem, you just move it to another part of your infrastructure. Most of the time, the shared data you want to store on the filesystem is written once and read often (or never). For this use case, S3 is ideal, if you can live with *eventual consistency*. In other words, it might take a little while before your update on an object becomes visible for reading.

www.it-ebooks.info

You can store the *nonfilesystem-style data* somewhere else. This is session information or other data that needs to be shared by several instances, for example. In the next chapter we will look at AWS solutions like SimpleDB and SQS. For now, we'll just move it to the database.

Then we are left with the *database*. As said before, the problem with a relational database when trying to scale is the join, or join-like operations, which are the most expensive ones. Of course, you can handle many joins if you replicate the database and perform these operations on one or more slaves, but the joins stay—they are not removed. The most radical way to scale the database without getting rid of joins is to minimize their execution. The easiest and most versatile way is caching. The popular choice for caching is Memcached.

Modern web application development frameworks make it very easy to utilize these techniques. Rails is not any different, as you'll see. Figure 3-2 shows a high-level view of the different components of Kulitzer, our Rails sample application.

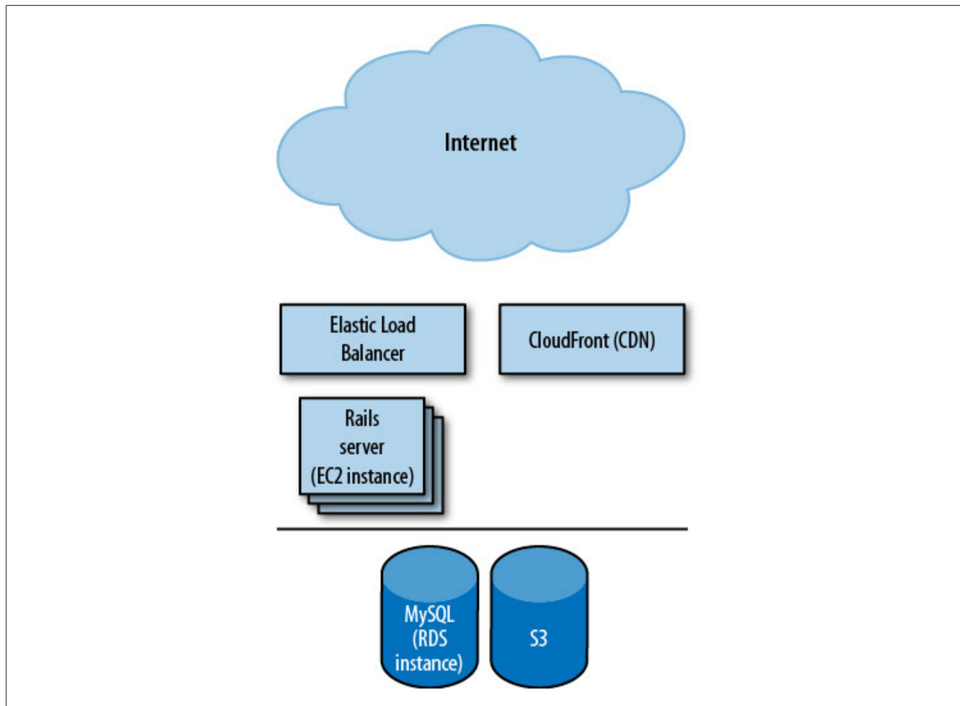


Figure 3-2. Kulitzer architecture

You will also see that RDS, the MySQL service provided by Amazon, can easily be scaled up and down if needed (up to a certain limit, of course).

www.it-ebooks.info

Table 3-1. Preparing to scale: options for sharing data

Presentation tier	Filesystem-style data	Nonfilesystem-style data
CDN, e.g., CloudFront over S3	S3	Database
	Database	SimpleDB
	SAN, NAS	SQS
	NFS, SMB (not very scalable)	

Setting Up the Tools

Before we start diving into these scaling techniques, it is a good idea to install some tools. Download the command-line tools for the following:

- ELB
- Auto Scaling
- CloudWatch

Extract the compressed files and add the bin directories to your `initaws` file as we did in Chapter 2.

S3 for File Uploads

When you want to store user content to S3, you have two aspects to consider:

- *Latency*. S3 is accessed through a web interface, which is considerably slower than disk (EBS).
- *Eventual consistency*. Even if the content is uploaded, it is not guaranteed to be available immediately, but it will be there eventually.

For these reasons, we need to find a way to accept the user content but offload the actual upload to S3 to a separate process/thread. We don't want to keep the user waiting for this work to complete. She will be notified that it might take a while and will automatically see the content when available. For big workloads, you might consider Amazon SQS, but that is a bit too much at this moment. We'll search for our solution in the framework we use, Ruby on Rails, in the case of the Kulitzer example to follow.

User Uploads for Kulitzer (Rails)

We chose to use Rick Olson's `attachment_fu` for handling user uploads. It does S3 out of the box, so we don't need to modify the plug-in to implement the actual upload. But it doesn't do this outside of the user thread, so we need to do some extra work there. Of course, we are not the only ones with this challenge, and we found that people were using a plug-in called `Bj` (BackgroundJob) in combination with `attachment_fu` to handle

www.it-ebooks.info

S3 file uploads. Ara Howard's `Bj` uses the database to create jobs, which are run one at a time, outside of the app server's process.

So with this, the user content is uploaded to S3. The location (URL) given to the actual assets is the S3 location. But we want to serve this content from CloudFront. We know all our images come from the same CloudFront distribution, so we can implement this quickly by overriding the `image_tag` method of the AssetTagHelper in `config/initializers/image_tag.rb`. In this method, the S3 URL is replaced with a URL from one of the domains (randomly chosen) that points to the CloudFront distribution. It might not be the most beautiful solution, but it is easy to understand and it works. If we need something more elaborate, we will probably have to move from `Bj` to Amazon SQS:

```
module Kulitzer
  module CloudFrontImageTag

    S3_PREFIX = 'http://s3.amazonaws.com/media.kulitzer.com'
    CF_PREFIX = 'http://media%d.kulitzer.com'

    def image_tag(source, options = {})
      # For production, re-write url to use CloudFront
      if Rails.env.include?('production') && source.starts_with?(S3_PREFIX)
        # re-write to use a cloudfront url eg
        # http://s3.amazonaws.com/media.kulitzer.com/foo.jpg becomes
        # http://mediaZ.kulitzer.com/foo.jpg where Z = 0..3

        cf_host = CF_PREFIX % rand(4)
        source = source.sub(S3_PREFIX, cf_host)
      end

      super
    end
  end
end

ActionView::Base.send :include, Kulitzer::CloudFrontImageTag
```

Elastic Load Balancing

Wikipedia says that *load balancing* is "a technique to distribute workload evenly across two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, minimize response time, and avoid overload." And, Wikipedia notes, it can also "increase reliability through redundancy." To finish the description, Wikipedia says load balancing "is usually provided by a dedicated program or hardware device."

Amazon *Elastic Load Balancing* (ELB) is all of the above. It distributes load evenly across availability zones and across instances within those zones. ELB checks the health of the instances and will not route traffic to unhealthy instances. You have the ability to use something called *sticky sessions*, which can force a particular session to one instance.

www.it-ebooks.info

You would need this in case the instances keep session data in a nonshared location such as local memory.

ELB is not a dedicated program or a hardware device; it is a load-balancing service. As a service, it can automatically scale its capacity depending on incoming traffic. As a result, an ELB is not referenced by an IP address, but by a fully qualified domain name. It's been said that an ELB scales best with slowly increasing/decreasing traffic, but in our experience spikes are handled quite well.

Layar is a high-profile augmented reality (AR) application for smartphones like Android and iPhone. The ongoing traffic is reasonable, but not extreme; there are just not very many AR-capable smartphones available yet (with camera, GPS, compass, and accelerometer). However, the Layar application is magic, and phone manufacturers like Samsung and Motorola use Layar to sell their latest devices.

Motorola especially is determined to sell as many Android smartphones as possible, and it often buys primetime television advertising slots. One of its ads focuses solely on Layar, and it was aired during Oprah Winfrey's popular TV show. We saw a spike in the Layar servers' traffic that was extreme, to say the least. We didn't have autoscaling yet, so our two instances were hit hard. We went from around 30 requests per second to over 800 requests per second. Any more was just not possible.

This was the first serious production test of the Layar infrastructure. It was a valuable lesson, especially with the other, even bigger events on the horizon.

Now that we are using S3 for file uploads, we are almost ready for load balancing, as we can't rely on instance storage anymore. There is one other thing we have to make sure we don't do, and that is use local files. Rails can store sessions in file storage. The default storage mechanism for sessions in Rails 2.x is CookieStore, and this is fine because it stores the session object in the client. Same for the alternative ActiveRecordStore (or "one of its derivatives," as the guide tells us) that we use for Kulitzer, which stores the session in the database.

Creating an ELB

Creating an ELB is straightforward, and you can use either the AWS Console or the command-line tools. Simply give the ELB a name (which will be part of the domain name you use for the ELB), tell it how to handle traffic (protocols to handle, such as HTTP, HTTPS, TCP, and how to route load balancer ports to other or the same ports in the instances), configure the health check (thresholds to decide if an instance is healthy or not), and add your zones and instances. As you are used to by now, the Console gives you a wizard to create an ELB (Figure 3-3).

www.it-ebooks.info

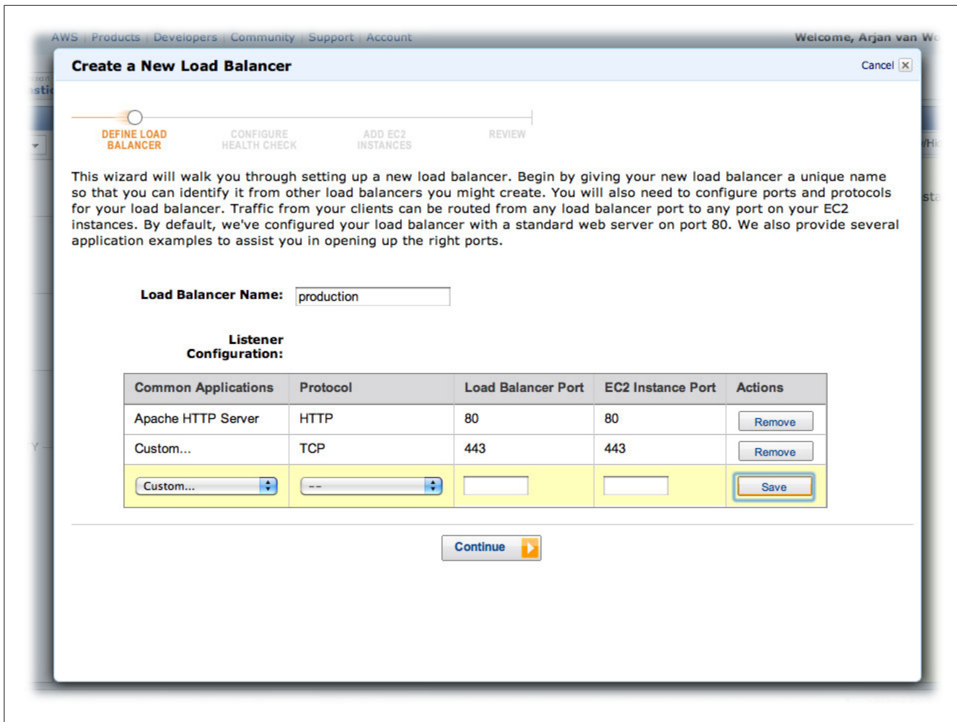


Figure 3-3. Create an Elastic Load Balancer

You can also create the ELB with the following commands:

```
$ elb-create-lb production \
        --availability-zones us-east-1b \
        --listener "protocol=HTTP, lb-port=80, instance-port=80" \
        --listener "protocol=TCP, lb-port=443, instance-port=443"

$ elb-configure-healthcheck production \
        --target "HTTP:80/" \
        --interval 30 \
        --timeout 2 \
        --healthy-threshold 6 \
        --unhealthy-threshold 2

$ elb-register-instances-with-lb production \
        --instances i-29184c43
```


www.it-ebooks.info

ELB and HTTPS

This particular load balancer routes HTTP traffic. It does not use the HTTPS feature of ELB, it just routes port 443 TCP traffic. This way, we handle the HTTPS on the instances.

You can also have the load balancer handle HTTPS traffic, but the communication between ELB and instances then remains unencrypted HTTP. You can't secure this 100% yet through the use of something similar to security groups, but if you make use of this, it does give you a couple of advantages, including the following:

- Extra load of HTTPS traffic is on the ELB, not on the instance. And the load of the ELB is taken care of as part of the service; you don't need to pay more for this.
- It is easier to set up and maintain.
- You can have session stickiness on HTTPS sessions.

Setting this up requires some IAM command magic to add your certificates, and a slightly different invocation of the ELB commands. Here we add two different wildcard certificates: **.publitas.nl* and **.publitas.com*. We pass the private key, public key certificate, the certificate chain file, and the name of the certificate to `iam-servercertupload`. Then we create the two load balancers from the command line:

```
$ iam-servercertupload -k _.publitas.nl.key \
    -b _.publitas.nl.crt -c gd_bundle.crt -s _publitas_nl
$ iam-servercertupload -k _.publitas.com.key \
    -b _.publitas.com.crt -c gd_bundle.crt -s _publitas_com

$ iam-servercertlistbypath
arn:aws:iam::075493990647:server-certificate/_publitas_com
arn:aws:iam::075493990647:server-certificate/_publitas_nl

$ elb-create-lb publitas-nl \
    --availability-zones eu-west-1a,eu-west-1b \
    --listener "protocol=HTTP, lb-port=80, instance-port=80" \
    --listener "protocol=HTTPS, lb-port=443, instance-port=80,
      cert-id=arn:aws:iam::075493990647:server-certificate/_publitas_nl"

$ elb-create-lb publitas-com \
    --availability-zones eu-west-1a,eu-west-1b \
    --listener "protocol=HTTP, lb-port=80, instance-port=80" \
    --listener "protocol=HTTPS, lb-port=443, instance-port=80,
      cert-id=arn:aws:iam::075493990647:server-certificate/_publitas_com"
```

When you create the load balancer from the command line, you need to specify the availability zones where your instances will run. In the AWS Console, the availability zones of the instances you choose will be added automatically. If you don't choose any instances, one availability zone will be chosen for you by default.

www.it-ebooks.info

If you work with ELBs from the Console, it appears as if you can only register an instance to one load balancer. It might be a mistake in the Console, but you can register instances to multiple load balancers at the same time. You can do this explicitly through the command line, or have Auto Scaling register multiple ELBs at the same time.

If you want to serve multiple SSL domains (actually, if you need multiple SSL certificates) through ELBs, you can't use the HTTPS feature of the load balancer. You can set up an ELB for every certificate, all pointing to the same instances.

You can clone your existing instance and make it work in a different availability zone. If you add this new instance to the load balancer, you have a nice load-balanced server farm for your application. This infrastructure is very close to our traditional infrastructures. There are many solutions to the problem of deploying an application on several servers; we use Capistrano in this case. With a little bit of work, we can scale our application up and down.

Because ELB does support load balancing on the HTTP level and TCP level, you are not restricted to balancing your web traffic. The default listeners are all web or application servers, but you can also use an ELB to set up a server farm for IMAP or POP, for example. All the principles we discussed still apply, of course.

Difficulties with ELB

Currently, there are a number of challenges with the ELB service. One of the most irritating, but logical, is that DNS does not allow CNAME records for root domains. In other words, you cannot point `kulitzer.com` to `production-1188255634.us-east-1.elb.amazonaws.com`. Some people think the only good domain name for an app is the shortest one possible, in our case `kulitzer.com`. But the hosts of most large sites rewrite their domains to the www alternative, probably because a service scales much easier than a device. You can use the rewrite, or forwarding, feature of your domain name registrar. We often use one of the app servers with an EIP address to do this rewriting. With Apache, you can add such a rewrite with `mod_rewrite`. We rewrite every domain name we don't want to `www.kulitzer.com` like this:

```
RewriteEngine              on
RewriteCond %{HTTP_HOST}   !^www\.kulitzer\.com [NC]
RewriteCond %{HTTP_HOST}   !^$
RewriteRule ^/(.*)         http://www.kulitzer.com/$1 [L,R]
```

Changing the configuration of a running ELB is limited. You can't add or remove different connections, for example. For this reason, just always do HTTP and HTTPS, if you have the proper certificates. The chances are small that you will want to balance other traffic later in the life of a load balancer, but a mistake with HTTP is inconvenient.

www.it-ebooks.info

Auto Scaling

The cloud's biggest promise is elasticity: running with just the amount of computing resources your workload demands at any given time. Of course, services like SimpleDB, SQS, and CloudFront scale automatically. But for some applications, you need to be able to dynamically scale EC2 capacity, or instances. And that is what Auto Scaling does. It allows you to define conditions for increasing or decreasing the number of instances.

According to AWS, "Auto Scaling is particularly well suited for applications that experience hourly, daily, or weekly variability in usage." This is exactly what we need for Kulitzer. At peak hours, we want to increase our capacity to handle the load, and when it's not so busy, we want to run idle.

You can also use Auto Scaling to make your application more *resilient*. Say you use Auto Scaling to monitor one instance; if the instance dies, Auto Scaling can launch a new one automatically. This same technique can be applied to multiple instances distributed over different availability zones. We basically apply elasticity in case of failure, automatically resizing the group to its required size.

Auto Scaling is designed to work on its own, but can be easily integrated with an ELB. You can't use metrics of the load balancer as conditions for scaling. But it is convenient that the autoscaling group handles registration of instances with the load balancer.

Usually, it takes several minutes for an instance to launch and provision itself. Because demand cannot be met immediately, extreme spikes are more difficult to handle than gradual changes in demand. But you can tune the triggers for scaling so that spiky conditions can be handled, more or less.

After Oprah introduced Layar, we knew we should plan for these spikes. We set up autoscaling and tuned the triggers so that we would scale up relatively early and scale down relatively late.

Not everyone at Layar expected another Oprah, and we were already quite surprised to have had to handle that spike. But in May 2010, the inevitable happened. The same ad was aired in primetime—not on a talk show, but during the fifth game of the NBA finals.

Within an hour, we scaled from the two large instances we normally run to 16 large instances. The database choked (again) and we couldn't handle any more traffic. But compared to our Oprah moment, we were able to serve many more users for just a few dollars.

Setting Up Auto Scaling

Auto Scaling is a little complex. There are *autoscaling groups* that hold the instances. You have *launch configurations* that determine which instance is launched. And there

www.it-ebooks.info

are *alarms*, borrowed from CloudWatch, that determine when a scaling action—called a *policy*—should take place. Policies simply specify that instances should be launched or terminated. All of these are dependent, making it difficult to know the order in which they can or should be changed. Working on a live Auto Scaling setup can be a bit scary. Don't be afraid, though—all Auto Scaling commands are quite verbose (not all EC2 commands are this "talkative") and it is difficult to do things you don't intend. But there is nothing better than testing a properly set up autoscaling group by terminating an instance and seeing a new instance taking its place.

Launch configuration

A launch configuration determines what kind of instance is being launched. It sets the AMI and the type of instance that is going to be created. Here we immediately run into the most important change in our application. We need to create self-provisioning instances; upon launch, an instance needs to have or get the right version of the application.

As you don't want to change images all the time, you probably want to devise a way of passing enough information to the instance for provisioning. Depending on your way of deploying an application, you can specify the branch (Subversion) or commit (GIT) in a place where the instance can read it. You can choose to store this in a public object in S3, as the rest of the information is safely stored on the image itself. If you want even more freedom, you can specify a complete provisioning script in S3 that is executed by the instance upon startup.

With your image ready, specify the launch configuration by executing the following command:

```
$ as-create-launch-config app-server-launch-config-1 \
    --image-id ami-6e1deb07 \
    --instance-type c1.medium \
    --group web
```

Autoscaling group

An autoscaling group must have a launch configuration, which is why you created one first. Now that we have one, we can create an autoscaling group. Depending on how you specify the autoscaling group, you might be launching instances immediately.

For Kulitzer, the autoscaling group will control between two and six instances behind an ELB. We want at least two instances to add redundancy, but the database cannot handle more than six instances at this moment. The instances will be distributed over two availability zones to make our application more resilient. Remember that availability zones are physically separated, forcing instances to use different cooling, racks, and blade servers. And we force the autoscaling group to wait a bit after scaling (out or in) by using the `cooldown` option:

```
$ as-create-auto-scaling-group app-server-as-group-1 \
    --launch-configuration app-server-launch-config-1 \
```


www.it-ebooks.info

```
    --availability-zones us-east-1c,us-east-1d \
    --min-size 2 \
    --max-size 6 \
    --default-cooldown 120 \
    --load-balancers production
```

> This group will immediately launch two instances. If you don't want that because you want to specify your triggers first, you can set `--min-size` to 0 and change it to the required size later. If your group does not have running instances, it is impossible to initiate scaling activities. The first scaling activity has to be manual in that case.

With the configuration done so far, this group will not really scale automatically, but it will make sure you keep the minimum you specified. You can test this by terminating one of the instances.

Sometimes it might happen that the request to start a new instance cannot be met, for example due to capacity problems. If you specified multiple availability zones in the autoscaling group, it will launch an instance in another availability zone. The autoscaling group will later try to rebalance the instances evenly over the availability zones. If your autoscaling group is hooked up with an ELB, instances will always be removed from the load balancer before being terminated. This is to ensure that the load balancer's service is not degraded.

Autoscaling (alarms and policies)

To make this autoscaling group really scale automatically, we need a way to monitor the group and execute a scaling policy. This used to be done with triggers (part of the Auto Scaling API), but it has changed to policies. The actual triggering has moved to CloudWatch, where metric alarms can execute a policy. The alarms determine when to take action, the actions are specified in policies, the autoscaling group defines the boundaries (min, max, availability zones), and the launch configuration defines which instances are created.

In the Kulitzer example, we will create two different policies: one for scaling up and one for scaling down. Each will be triggered by a different alarm, also shown here:

```
$ as-put-scaling-policy app-server-scale-UP-on-CPU \
    --auto-scaling-group app-server-as-group-1 \
    --type ChangeInCapacity \
    --adjustment 2 \
    --cooldown 300

$ mon-put-metric-alarm alarm-app-server-scale-UP \
    --alarm-actions arn:aws:autoscaling:us-east-1:205414005158:scalingPolicy:...
    --metric-name CPUUtilization \
    --unit Percent \
    --namespace AWS/EC2 \
    --statistic Average \
```


www.it-ebooks.info

```
    --dimensions="AutoScalingGroupName=app-server-as-group-1" \
    --period 60 \
    --evaluation-periods 2 \
    --threshold 60 \
    --comparison-operator GreaterThanThreshold

$ as-put-scaling-policy app-server-scale-DOWN-on-CPU \
    --auto-scaling-group app-server-as-group-1 \
    --type ChangeInCapacity \
    --adjustment=-2

$ mon-put-metric-alarm alarm-app-server-scale-UP \
    --alarm-actions arn:aws:autoscaling:us-east-1:205414005158:scalingPolicy:...
    --metric-name CPUUtilization \
    --unit Percent \
    --namespace AWS/EC2 \
    --statistic Average \
    --dimensions="AutoScalingGroupName=app-server-as-group-1" \
    --period 60 \
    --evaluation-periods 5 \
    --threshold 20 \
    --comparison-operator LessThanThreshold
```

It is important for the number of instances you scale up or down to be a multiple of the number of availability zones your autoscaling group utilizes in case your autoscaling group is connected to an ELB. The load balancer will distribute the traffic evenly over the active zones.

When we create the alarm, we pass the policy identifier returned by the `as-put-scaling-policy` command. There are different metrics we can check; in this case, we scale up when the `CPUUtilization` is more than 60% on average over all the instances of the given image for two consecutive periods of one minute each. Similarly, for scaling down, we check that the `CPUUtilization` on average is less than 20% for five consecutive periods of one minute.

Policies and metric alarms are very important for operating the autoscaling group. It is important to understand your application in order to choose which measure to use. You can tune the alarms based on your needs and requirements.

You can't make assumptions about which instances are terminated when the autoscaling group scales down. But if you scale regularly, the chances are good that the turnover of the physical hardware is quite high. This makes your application much more robust, because the chances of hardware failure increase with the age of deployment.

The steps to configure autoscaling, then, are the following:

1. Create a launch configuration, specifying the AMI and type of instances you want.
2. Create an autoscaling group, passing the above launch configuration, the availability zone(s), the minimum and maximum number of instances, and load balancer(s), if any.

www.it-ebooks.info

3. Create policies that represent scaling actions to take, like launching new instances or terminating existing ones.
4. Create alarms that trigger the execution of the above policies under certain conditions.

Semiautoscaling

Sometimes you don't want to scale based on the state of the group itself, but on a particular day and time. This might at first seem inefficient, because it's not directly triggered by a change in load. But if you know the door to the shop opens at 09:00 and you're selling the iPhone 7, you do want to preheat your server farm.

Instead of scaling up and down based on `CPUUtilization`, we want to make sure we have 24 instances running between 08:30 and 18:00 until 23 December 2011. On Christmas Eve, everyone is doing something else, so we can terminate this special group. We can do that like this:

```
$ as-put-scheduled-update-group-action christmas-app-server-scale-UP \
    --auto-scaling-group app-server-as-group-1 \
    --desired-capacity 24 \
    --time 2010-12-23T08:30:00Z

$ as-put-scheduled-update-group-action \
        christmas-app-server-scale-DOWN-on-christmas-eve \
    --auto-scaling-group app-server-as-group-1 \
    --desired-capacity 0 \
    --time 2010-12-24T20:00:00Z
```

Auto Scaling in Production

AWS Auto Scaling is great, and when used in combination with ELB it is a powerful tool in bringing down your costs by optimizing resource usage. In production, however, it is a bit difficult to handle. You can perform most common tasks easily if you know how, and all destructive activities are protected by warnings and dialog-style questions. The change we most often have to apply is changing a launch configuration because the image has changed.

Pausing Auto Scaling

When working on the autoscaling groups or instances in the group, autoscaling activities are unwelcome guests. In these cases, you can instruct AWS to leave your autoscaling group alone. They call this *suspend* and *resume*, and you can do it like this:

```
$ as-suspend-processes app-server-as-group-1
$ as-resume-processes app-server-as-group-1
```


www.it-ebooks.info

Replacing the launch configuration

If the image has changed for any reason, you will want to replace the launch configuration. The launch configuration only states what is to be done in the future, so it doesn't change anything that is currently in production. We always rotate instances when changing the launch configuration:

```
# first create a new launch configuration, then change the auto scaling group
$ as-create-launch-config app-server-launch-config-2 \
    --image-id ami-c503e8ac \
    --instance-type c1.medium \
    --group web
$ as-update-auto-scaling-group app-server-as-group-1 \
    --launch-configuration app-server-launch-config-2


# now replace the instances gracefully
$ as-terminate-instance-in-auto-scaling-group i-433e1829 -D

# and if the new instances is 'In Service' terminate the other
$ as-terminate-instance-in-auto-scaling-group i-413e182b -D
```

`-D` indicates that we don't want to decrement the capacity of the autoscaling group by 1. The result is that a new instance will be launched to compensate for each terminated instance. If we wanted instead to decrement the capacity of the group, we would use the `-d` option, and a new instance would be launched. This is not, of course, what we want in this example.

Unfortunately, replacing instances this way terminates the old instance before registering the new one. This means you are temporarily running with one fewer instance. Make sure you wait until the new instance is registered, healthy, and In Service before terminating the next.

Changing the alarms

The alarms are potentially more dangerous to your production environment. By changing the alarms, you can force your autoscaling group to breach the thresholds and initiate scaling activities. However, it is not very difficult to change the alarms, though it is slightly different from the launch configuration. Instead of first creating a new alarm, you can just update the existing one:

```
$ mon-put-metric-alarm alarm-app-server-scale-UP \
    --period 300 \
    --evaluation-periods 2 \
    --threshold 30 \
    --comparison-operator LessThanThreshold
```

This command may seem a bit long, but you have to specify everything again.

www.it-ebooks.info

Changing the autoscaling group

If you are technically breaching, but running at minimum or maximum capacity, the autoscaling group does not launch or terminate instances. In our case, two medium instances is too much at idle time. If we decrease the minimum size of the autoscaling group, the remaining two instances will be terminated:

```
$ as-update-auto-scaling-group app-server-as-group-1 \
    --min-size 0
```

Decommissioning an autoscaling group

If you terminate an instance directly, the autoscaling group will probably just launch another one. If you use the command we showed for rotating instances, the same will happen. Set the minimum size of the autoscaling group to 0 and, if necessary, terminate the instances by hand:

```
$ as-update-auto-scaling-group app-server-as-group-1 \
    --min-size 0
$ as-terminate-instance-in-auto-scaling-group i-870620ed -d
$ as-terminate-instance-in-auto-scaling-group i-810620eb -d
$ as-delete-auto-scaling-group app-server-as-group-1
```

Scaling a Relational Database

It's amazing what the AWS team achieved with RDS in a little over a year since it was introduced. The team added the high-availability feature it calls multi-AZ, or multiavailability zone. And after that, it also added *read replicas*. With these new features, in addition to the existing scalability features, RDS is growing into a serious RDBMS service. You can easily scale up while minimizing downtime, and also scale out without too much hassle.

If your app gets to the point that you need to start scaling either up or out, it is a good idea to switch to multi-AZ if you don't run it already. If you have a simple RDS instance, you will degrade your service significantly while scaling, as you can expect to lose the ability to write and/or read. With multi-AZ RDS instances, your service is almost uninterrupted.

Scaling Up (or Down)

Scaling up is so easy it is almost ridiculous. The only drawback is that you have some downtime during the operation. If you don't have multi-AZ enabled, the downtime of your RDS instance could be several minutes, as you have to wait until a new instance is launched and fully functional. For multi-AZ RDS instances, you will experience some downtime as a failover is initiated after the slave has been scaled up (or down). This failover doesn't take more than a minute most of the time.

www.it-ebooks.info

If you initiate a scaling activity via the Console, make sure you enable Apply Immediately if you are in a hurry. If you don't, scaling will take place during the scheduled maintenance period (Figure 3-4).

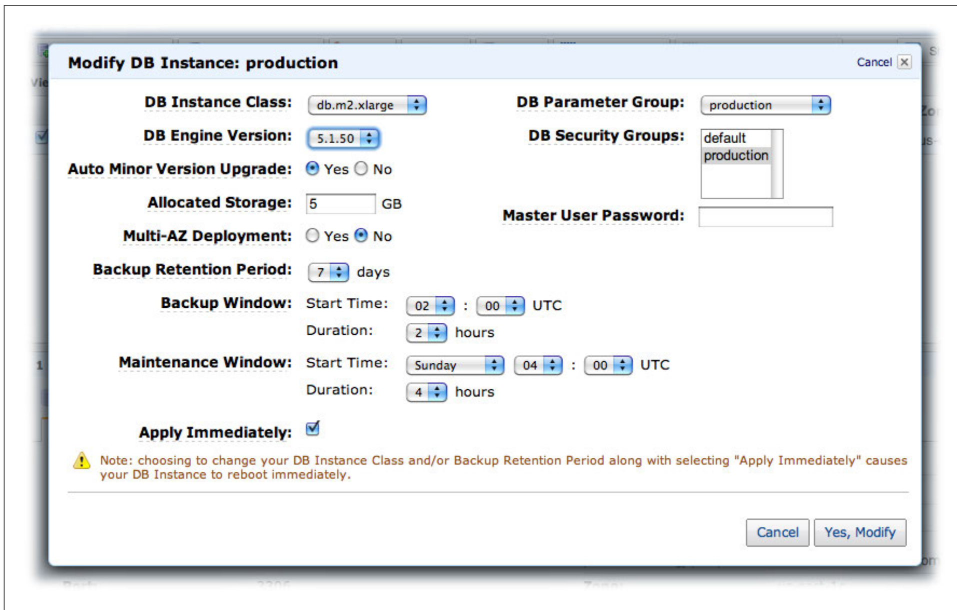


Figure 3-4. Modify the RDS instance (scaling up)

Scaling using the command-line tools is a two-step process. First scale, and then reboot:

```
$ rds-modify-db-instance production \
        --db-instance-class db.m1.xlarge --apply-immediately
$ rds-reboot-db-instance production
```

DB instance classes

Of course, every service in AWS uses a slightly different naming convention. The equivalent of EC2 Instance Type for RDS is called DB Instance Class. Luckily, the classes themselves are more or less similar to the types in EC2. The smallest possible RDS instance you can have is comparable to a small EC2 instance, for example, though we experience the performance as a bit more consistent with the RDS instance. Here are all the instance classes with their descriptions as AWS advertises them:

Small DB Instance
: 1.7 GB memory, 1 EC2 Compute Unit (1 virtual core with 1 ECU), 64-bit platform, moderate I/O capacity

Large DB Instance
: 7.5 GB memory, 4 ECUs (2 virtual cores with 2 ECUs each), 64-bit platform, high I/O capacity

www.it-ebooks.info

Extra Large DB Instance
: 15 GB memory, 8 ECUs (4 virtual cores with 2 ECUs each), 64-bit platform, high I/O capacity

High-Memory Extra Large DB Instance
: 17.1 GB memory, 6.5 ECUs (2 virtual cores with 3.25 ECUs each), 64-bit platform, high I/O capacity

High-Memory Double Extra Large DB Instance
: 34 GB memory, 13 ECUs (4 virtual cores with 3.25 ECUs each), 64-bit platform, high I/O capacity

High-Memory Quadruple Extra Large DB Instance
: 68 GB memory, 26 ECUs (8 virtual cores with 3.25 ECUs each), 64-bit platform, high I/O capacity

Scaling Out

You can scale out a relational database in two different ways:

- Using read-only slaves (*read replicas* in AWS)
- Sharding or partitioning

There are still some hard problems to solve, as sharding/partitioning has not been addressed yet with RDS. Master-slave type scaling is available, though. A slave, or read replica, is easily created from the Console (Figure 3-5). The only requirement on the master RDS instance is that backups are not disabled by setting the backup retention period to 0. Currently, you can have up to five read replicas that you have to launch one by one. Amazon is working on the ability to launch multiple replicas at once, but that is not yet available.

On a multi-AZ RDS instance, launching a read replica goes unnoticed. A snapshot is taken from the standby, the replica is launched, and when it is ready, it starts to catch up with the master. For a normal RDS instance, there is a brief I/O suspension in the order of one minute. AWS advises to use the same instance classes, as differing classes may incur replica lag. With read replicas, you basically introduce eventual consistency in your database (cluster).

The read replica mechanism uses MySQL's native, asynchronous replication. This means replicas might be lagging behind the master as they try to catch up with writes. The interesting thing about this is that multi-AZ RDS instances apparently use another, proprietary type of synchronous replication.

www.it-ebooks.info

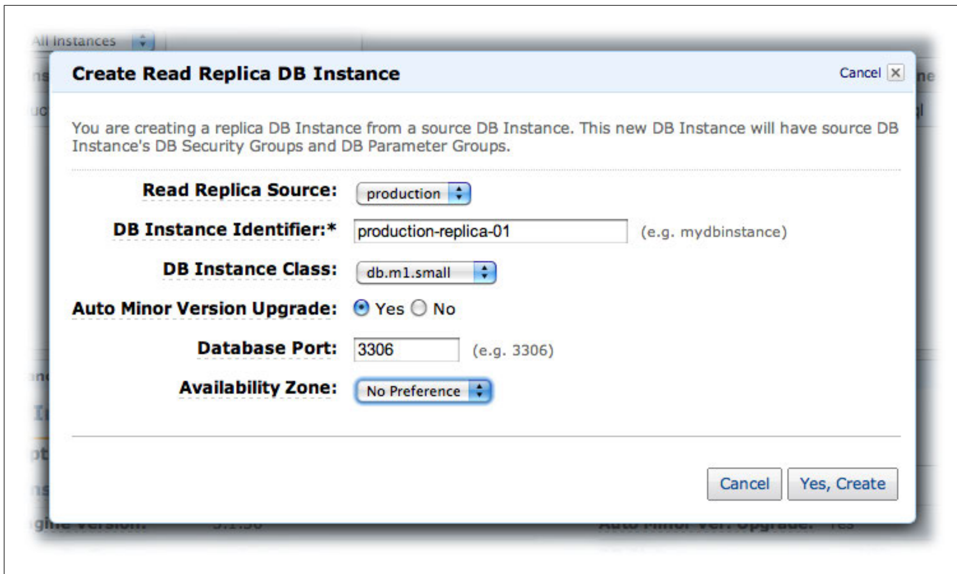


Figure 3-5. Create a read replica

Storage engine

The default storage engine with RDS is InnoDB, but you are free to choose another, like the popular MyISAM. It is important to realize that read replicas on nontransactional storage engines (like MyISAM) require you to freeze your databases, as the consistency cannot be guaranteed when snapshotting. But if you use InnoDB, you are safe, and the only thing you have to do is fire up a new read replica.

Tips and Tricks

RDS is MySQL, but you don't have as much control as you would with MySQL. There are certain peculiarities in RDS' MySQL implementation that can cause a lot of frustration.

Disk is slow

A disk is always slower than memory. If you run your own MySQL using local disks, that's slow as well. But using disk-based operations in RDS is just horrible. Minimizing disk usage means implementing proper indexes, something you always want to do. Other operations that require fast disk access are views. For those experienced with Oracle, this is a huge surprise, but the views in MySQL are not that mature yet.

www.it-ebooks.info

Slow log

We always enable slow query logging. Sometimes the slow query grows too much. You can access the slow log through the `slow_log` table in the `mysql` database. But you can't just truncate; you have to use the procedure `rds_rotate_slow_log` by executing the following command in your MySQL client:

```
> CALL rds_rotate_slow_log
```

Storage

RDS storage is independent of RDS instance classes. Every class can have from 5 GB to 1 TB of storage associated. Scaling up the storage is easy, and you can do it using the Console. It does require a reboot. On the other hand, scaling down the storage is impossible.

Elastic Beanstalk

As we described in this chapter, AWS offers a number of services that help you scale your application. There are numerous ways to configure the different components, such as Auto Scaling and Elastic Load Balancing. AWS gives you a lot of power, but sometimes you just want a quicker way to get up and running with your app.

And that is exactly what Elastic Beanstalk (EB) can do for you. At this moment EB is Java-only, although this should be just the start. EB works with Tomcat as the Java app server, and the only thing you have to do is give it a WAR (Web application ARchive).

We don't have our own WAR, so we chose to run the provided sample application. After signing up for EB, we were able to launch this application successfully (see Figure 3-6).

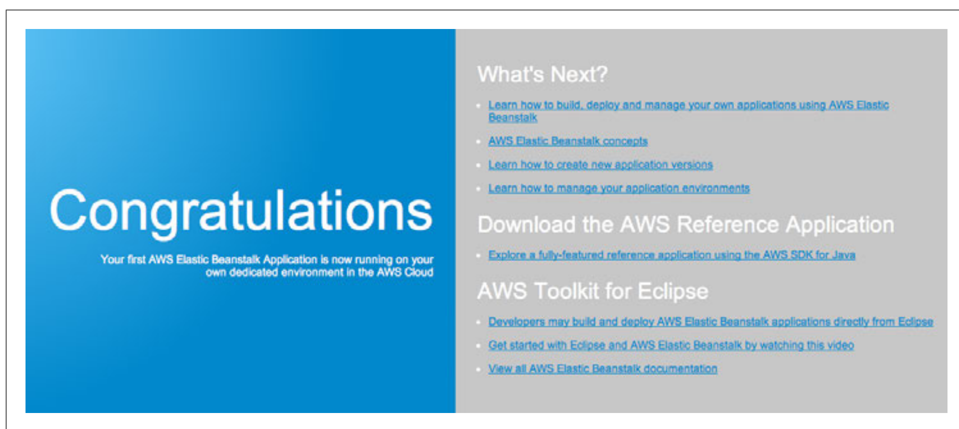


Figure 3-6. Elastic Beanstalk sample application

www.it-ebooks.info

So what happened here? The default EB setup consists of one micro instance behind an ELB part of an autoscaling group. A security group was created as well. These AWS components are given defaults to operate with. The autoscaling group, for example, has a minimum of one instance and maximum of four.

The supplied AMI when launching a Beanstalk sample application is 32-bit. This means that you can only choose between micro, small, and medium instances. If you create a new application, you can choose the architecture type of the instance you want to use for your environment, either 32-bit or 64-bit.

For production environments with sustained traffic, micro instances are probably not sufficient. If you expect traffic, you will definitely want to choose either small or medium. If you do this, take into account that you will have much more memory, but if you want to utilize this memory, you have to explicitly change the Java Virtual Machine (JVM) settings in the Container tab of the Edit Configuration dialog (Figure 3-7).

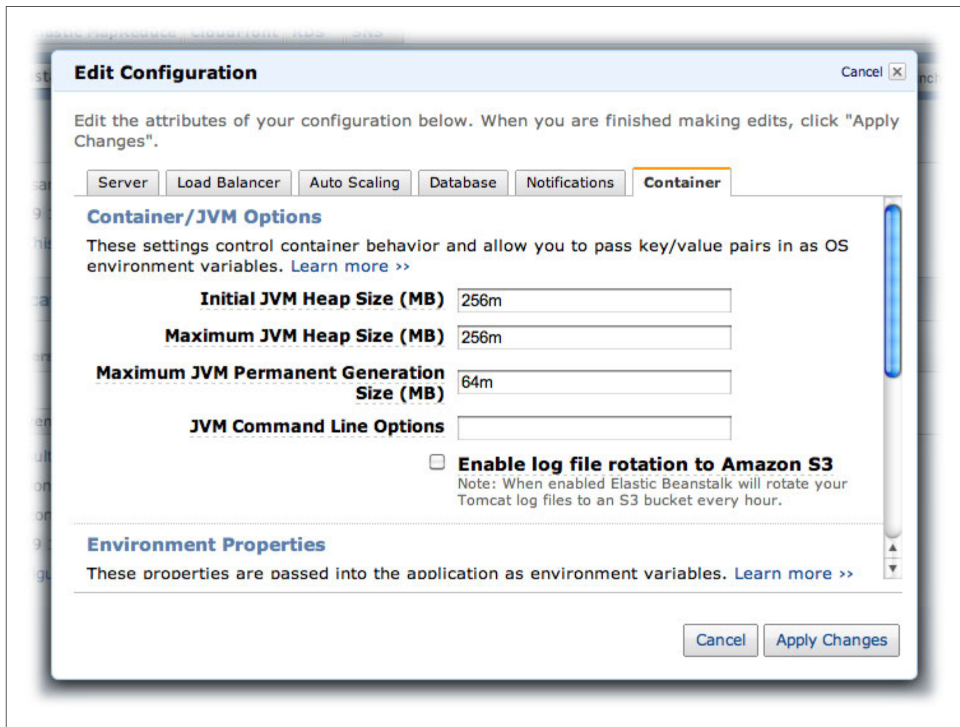


Figure 3-7. JVM settings in Elastic Beanstalk

Most of the settings we have seen in this chapter can be configured in the Elastic Beanstalk section of the AWS Console. This is nice, as we don't have Auto Scaling in the Console yet, for example. Another great feature they integrated with EB is Notifications. This allows you to get emails (using Amazon SNS) about events in your infrastructure, such as when instances are launched or terminated.

www.it-ebooks.info

There is one more feature worth mentioning, and that is deployment. EB allows you to manage versions of your application. Deployments happen on a running EB cluster (for lack of a better name). If things do not go as planned, you can easily choose another version to deploy, for example when you want to do a rollback. This might just be the killer feature of Elastic Beanstalk.

Even though this service has just been released, it will surely prove useful in helping you get up and running quickly. And if you outgrow the Elastic Beanstalk way of deploying your application, you can start to take over one step at a time.

In Short

In this chapter, we started taking advantage of the elasticity property of AWS infrastructures. On one hand you can scale up manually, changing to bigger instances. However, you can also scale *out* by using more instances, and AWS provides ways of automating this. Before you can set up your infrastructure to scale out, though, you need a way to put your data somewhere so it can be shared. We explored different ways to do this, and showed an example of using S3 for user uploaded data.

The two most important new components we introduced in this chapter are ELB and Auto Scaling. You can use these components together to set up an autoscalable infrastructure. By specifying a few rules, you can determine when your infrastructure will scale out by launching new instances, or scale in by terminating instances when load is reduced again, for example. We also discussed using Auto Scaling without an ELB to make your application more resilient by monitoring an instance and launching a new one when it fails.

We then focused on scaling relational databases. We showed how easy it is to scale an RDS instance up and down by choosing another instance class. For scaling out, we have the option of read replicas. In both cases, using multi-AZ RDS reduces or removes downtime when you make modifications to the RDS machines.

Finally, we gave a brief overview of the Elastic Beanstalk tool, which helps you set up a scalable application very quickly using EC2, Auto Scaling, ELB, S3, and SNS.

www.it-ebooks.info

CHAPTER 4

Decoupling with SQS, SimpleDB, and SNS

Take a moment to look back at what we've achieved so far. We started with a simple web application, with an infrastructure that was ready to fail. We took this app and used AWS services like Elastic Load Balancing and Auto Scaling to make it resilient, with certain components scaling out (and in) with traffic demand. We built an infrastructure (without any investment) that is capable of handling huge traffic at relatively low operational cost. Quite a feat!

Scaling beyond this amount of traffic requires more drastic changes in the application. We need to decouple, and for that we'll use Amazon SimpleDB, Amazon Simple Notification Service (SNS), and Amazon Simple Queue Service (SQS), together with S3, which we have already seen in action. But these services are more versatile than just allowing us to scale. We can use the decoupling principle in other scenarios, either because we already have distinct components or because we can easily add functionality.

In this chapter, we'll be presenting many different use cases. Some are already in production, others are planned or dreamed of. The examples are meant to show what you can do with these services and help you to develop your own by using code samples in various languages. We have chosen to use real-world applications that we are working with daily. The languages are Java, PHP, and Ruby, and the examples should be enough to get you going in other languages with libraries available.

SQS

In the SQS Developer Guide, you can read that "Amazon SQS is a distributed queue system that enables web service applications to quickly and reliably queue messages that one component in the application generates to be consumed by another component. A queue is a temporary repository for messages that are awaiting processing."

www.it-ebooks.info

And that's basically all it is! You can have many writers hitting a queue at the same time. SQS does its best to preserve order, but the distributed nature makes it impossible to guarantee it. If you really need to preserve order, you can add your own identifier as part of the queued messages, but approximate order is probably enough to work with in most cases. A trade-off like this is necessary in massively scalable services like SQS. This is not very different from eventual consistency, as seen in S3 and (as we will show soon) in SimpleDB.

You can also have many readers, and SQS guarantees each message is delivered at least once. Reading a message is atomic—locks are used to keep multiple readers from processing the same message. Because in such a distributed system you can't assume a message is not immediately deleted, SQS sets it to invisible. This invisibility has an expiration, called *visibility timeout*, that defaults to 30 seconds. If this is not enough, you can change it in the queue or per message, although the recommended way is to use different queues for different visibility timeouts. After processing the message, it must be deleted explicitly (if successful, of course).

You can have as many queues as you want, but leaving them inactive is a violation of intended use. We couldn't figure out what the penalties are, but the principle of cloud computing is to minimize waste. Message size is variable, and the maximum is 64 KB. If you need to work with larger objects, the obvious place to store them is S3. In our examples, we use this combination as well.

One last important thing to remember is that messages are not retained indefinitely. Messages will be deleted after four days by default, but you can have your queue retain them for a maximum duration of two weeks.

We'll show a number of interesting applications of SQS. For Kulitzer, we want more flexibility in image processing, so we've decided to decouple the web application from the image processing. For Marvia, we want to implement delayed PDF processing: users can choose to have their PDFs processed later, at a cheaper rate. And finally, we'll use Decaf to have our phone monitor our queues and notify when they are out of bounds.

Example 1: Offloading Image Processing for Kulitzer (Ruby)

Remember how we handle image processing with Kulitzer? We basically have the web server spawn a background job for asynchronous processing, so the web server (and the user) can continue their business. The idea is perfect, and it works quite well. But there are emerging conversations within the team about adding certain features to Kulitzer that are not easily implemented in the current infrastructure.

Two ideas floating around are RAW images and video. For both these formats, we face the same problem: there is no ready-made solution to cater to our needs. We expect to have to build our service on top of multiple available free (and less free) solutions. Even though these features have not yet been requested and aren't found on any road map, we feel we need to offer the flexibility for this innovation.

www.it-ebooks.info

For the postprocessing of images (and video) to be more flexible, we need to separate this component from the web server. The idea is to implement a postprocessing component, picking up jobs from the SQS queue as they become available. The web server will handle the user upload, move the file to S3, and add this message to the queue to be processed. The images (thumbnails, watermarked versions, etc.) that are not yet available will be replaced by a "being processed" image (with an expiration header in the past). As soon as the images are available, the user will see them. Figure 4-1 shows how this could be implemented using a different EC2 instance for image processing in case scalability became a concern. The SQS image queue and the image processing EC2 instance are introduced in this change.

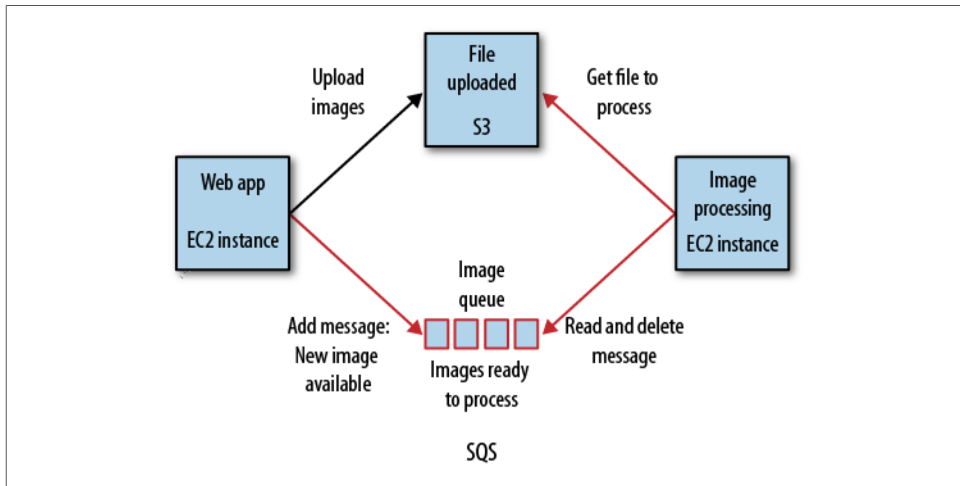


Figure 4-1. Offloading image processing

We already have the basics in our application, and we just need to separate them. We will move the copying of the image to S3 out of the background jobs, because if something goes wrong we need to be able to notify the user immediately. The user will wait until the image has been uploaded, so he can be notified on the spot if something went wrong (wrong file image, type, etc.). This simplifies the app, making it easier to maintain. If the upload to S3 was successful, we add an entry to the `images` queue.

SQS is great, but there are not many tools to work with. You can use the SQS Scratchpad, provided by AWS, to create your first queues, list queues, etc. It's not a real app, but it's valuable nonetheless. You can also write your own tools on top of the available libraries. If you work with Python, you can start the shell, load boto, and do the necessary work by hand.

www.it-ebooks.info

This is all it takes to create a queue and add the image as a queue message. You can run this example with `irb`, using the Ruby gem from RightScale `right_aws`. Use `images_queue.size` to verify that your message has really been added to the queue:

```
require 'rubygems'
require 'right_aws'

# get SQS service with AWS credentials
sqs = RightAws::SqsGen2.new("AKIAIGKECZXA7AEIJLMQ",
                            "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")

# create the queue, if it doesn't exist, with a VisibilityTimeout of 120 (seconds)
images_queue = sqs.queue("images", true, 120)

# the only thing we need to pass is the URL to the image in S3
images_queue.send_message(
  "https://s3.amazonaws.com/media.kulitzer.com.production/212/24/replication.jpg")
```

That is more or less what we will do in the web application to add messages to the queue. Getting messages from the queue, which is part of our image processing component, is done in two steps. If you only get a message from the queue—`receive`—SQS will set that message to invisible for a certain time. After you process the message, you can `delete` it from the queue. `right_aws` provides a method `pop`, which both receives and deletes the message in one operation. We encourage you *not* to use this, as it can lead to errors that are very hard to debug because you have many components in your infrastructure and a lot of them are transient (as instances can be terminated by Auto Scaling, for example). This is how we pick up, process, and, upon success, delete messages from the `images_queue`:

```
require 'rubygems'
require 'right_aws'

sqs = RightAws::SqsGen2.new("AKIAIGKECZXA7AEIJLMQ",
                            "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")

# create the queue, if it doesn't exist, with a VisibilityTimeout of 120 (seconds)
images_queue = sqs.queue("images", true, 120)

# now get the messages
message = images_queue.receive

# process the message, if any
# the process method is application-specific
if (process(message.body)) then
    message.delete
end
```

This is all it takes to decouple distinct components of your application. We only pass around the full S3 URL, because that is all we need for now. The different image sizes that are going to be generated will be put in the same directory in the same bucket, and are distinguishable by filename. Apart from being able to scale, we can also scale flexibly. We can use the queue as a buffer, for example, so we can run the processing app

www.it-ebooks.info

with fewer resources. There is no user waiting and we don't have to be snappy in performance.

> This is also the perfect moment to introduce *reduced redundancy storage*, which is a less reliable kind of storage than standard S3, but notifies you when a particular file is compromised. The notification can be configured to place a message in the queue we just created, and our post-processing component recognizes it as a compromised file and regenerates it from the original.
>
> Reduced redundancy storage is cheaper than standard S3 and can be enabled per file. Especially with large content repositories, you only require the original to be as redundant as S3 originally was. For all generated files, we don't need that particular security.
>
> Notification is through SNS, which we'll see in action in a little bit. But it is relatively easy to have SNS forward the notification to the queue we are working with.

Example 2: Priority PDF Processing for Marvia (PHP)

Marvia is a young Dutch company located in the center of Amsterdam. Even though it hasn't been around for long, its applications are already very impressive. For example, it built the Speurders application for the Telegraaf (one of the largest Dutch daily newspapers), which allows anyone to place a classified ad in the print newspaper. No one from the Telegraaf bothers you, and you can decide exactly how your ad will appear in the newspaper.

But this is only the start. Marvia will expose the underlying technology of its products in what you can call an API. This means Marvia is creating a PDF cloud that will be available to anyone who has structured information that needs to be printed (or displayed) professionally.

One of the examples showing what this PDF cloud can do is Cineville.nl, a weekly film schedule that is distributed in print in and around Amsterdam. Cineville gets its schedule information from different cinemas and automatically has its PDF created. This PDF is then printed and distributed. Cineville has completely eradicated human intervention in this process, illustrating the extent of the new industrialization phase we are entering with the cloud.

The Marvia infrastructure is already decoupled, and it consists of two distinct components. One component (including a web application) creates the different assets needed for generating a PDF: templates, images, text, etc. These are then sent to the second component, the InDesign farm. This is an OSX-based render farm, and since AWS does not (yet) support OSX, Marvia chose to build it on its own hardware. This architecture is illustrated in Figure 4-2.

www.it-ebooks.info

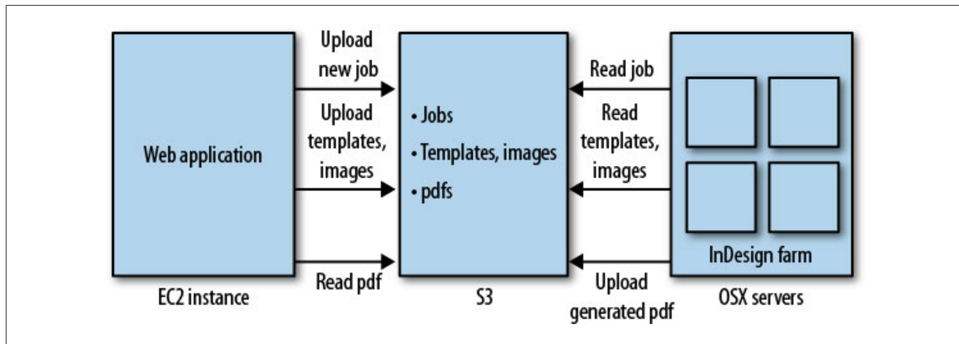


Figure 4-2. Marvia's current architecture

The two components already communicate, by sharing files through S3. It works, but it lacks the flexibility to innovate the product. One of the ideas being discussed is to introduce the concept of quality of service. The quality of the PDFs is always the same—they're very good because they're generated with care, and care takes time. That's usually just fine, but sometimes you're in a hurry and you're willing to pay something extra for preferential treatment.

Again, we built in flexibility. For now we are stuck with OSX on physical servers, but as soon as that is available in the cloud, we can easily start optimizing resources. We can use spot instances (see the Tip on page 35), for example, to generate PDFs with the lowest priority. The possibilities are interesting.

But let's look at what is necessary to implement a basic version of quality of service. We want to create two queues: high-priority and normal. When there are messages in the high-priority queue, we serve those; otherwise, we work on the rest. If necessary, we could add more rules to prevent starvation of the low-priority queue. These changes are shown in Figure 4-3.

Installing the tools for PHP

AWS has a full-fledged PHP library, complete with code samples and some documentation. It is relatively easy to install using PHP Extension and Application Repository (PEAR). To install the stable version of the AWS PHP library in your PHP environment, execute the following commands (for PHP 5.3, most required packages are available by default, but you do have to install the JSON library and cURL):

```
$ pear install pecl/json
$ apt-get install curl libcurl3 php5-curl

$ pear channel-discover pear.amazonwebservices.com
$ pear install aws/sdk
```


www.it-ebooks.info

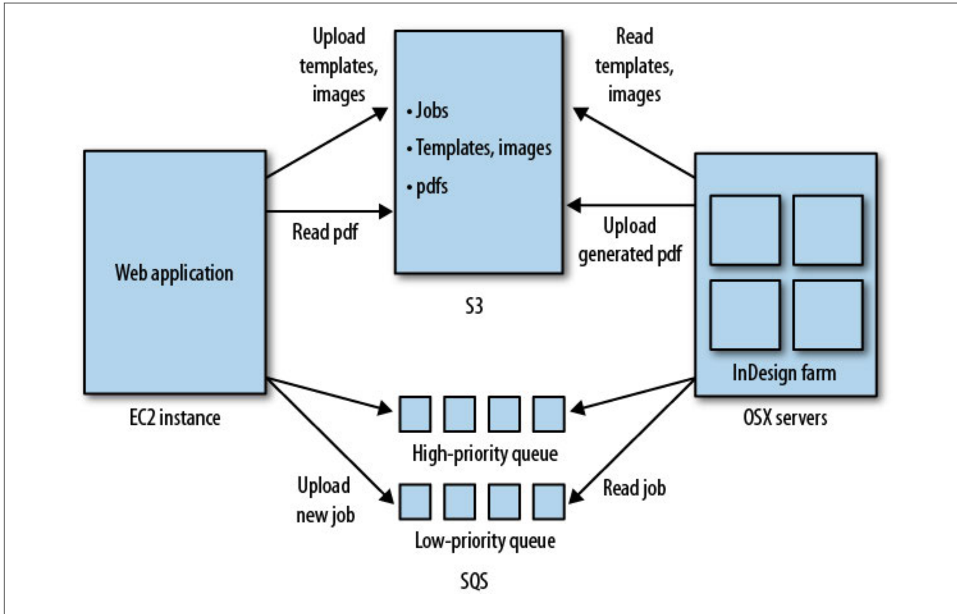


Figure 4-3. Using SQS to introduce quality of service

Writing messages

The first script receives messages that are posted with HTTP, and adds them to the right SQS queue. The appropriate queue is passed as a request parameter. The job description is fictional, and hardcoded for the purpose of the example. We pass the job description as an encoded JSON array. The queue is created if it doesn't exist already:

> We have used the AWS PHP SDK as is, but the path to `sdk.class.php` might vary depending on your installation. The SDK reads definitions from a config file in your home directory or the directory of the script. We have included them in the script for clarity.

```
<?php
    require_once( '/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');
    define('AWS_ACCOUNT_ID', '457964863276');

    # get queue name
    $queue_name = $_GET['queue'];

    # construct the message
    $job_description = array(
      'template' =>
```


www.it-ebooks.info

```
        'https://s3-eu-west-1.amazonaws.com/production/templ_1.xml',
      'assets' =>
        'https://s3-eu-west-1.amazonaws.com/production/assets/223',
      'result' =>
        'https://s3-eu-west-1.amazonaws.com/production/pdfs/223');
    $body = json_encode( $job_description);

    $sqs = new AmazonSQS();
    $sqs->set_region($sqs::REGION_EU_W1);

    $high_priority_jobs_queue = $sqs->create_queue( $queue_name);
    $high_priority_jobs_queue->isOK() or
        die('could not create queue high-priority-jobs');

    # add the message to the queue
    $response = $sqs->send_message(
        $high_priority_jobs_queue->body->QueueUrl(0),
        $body);

    pr( $response->body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```

Below, you can see an example of what the output might look like (we would invoke it with something like `http://<elastic_ip>/write.php?queue=high-priority`):

```
CFSimpleXML Object
(
    [@attributes] => Array
        (
            [ns] => http://queue.amazonaws.com/doc/2009-02-01/
        )

    [SendMessageResult] => CFSimpleXML Object
        (
            [MD5OfMessageBody] => d529c6f7bfe37a6054e1d9ee938be411
            [MessageId] => 2ffc1f6e-0dc1-467d-96be-1178f95e691b
        )

    [ResponseMetadata] => CFSimpleXML Object
        (
            [RequestId] => 491c2cd1-210c-4a99-849a-dbb8767d0bde
        )

)
```

Reading messages

In this example, we read messages from the SQS queue, process them, and delete them afterward:

www.it-ebooks.info

```php
<?php
    require_once('/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');
    define('AWS_ACCOUNT_ID', '457964863276');

    $queue_name = $_GET['queue'];

    $sqs = new AmazonSQS();
    $sqs->set_region($sqs::REGION_EU_W1);

    $queue = $sqs->create_queue($queue_name);
    $queue->isOK() or die('could not create queue ' . $queue_name);

    $receive_response = $sqs->receive_message( $queue->body->QueueUrl(0));

    # process the message...

    $delete_response = $sqs->delete_message( $queue->body->QueueUrl(0),
        (string)$receive_response->body->ReceiptHandle(0));

    $body = json_decode($receive_response->body->Body(0));
    pr( $body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```

The output for this example would look like this (we would invoke it with something like `http://<elastic_ip>/read.php?queue=high-priority`):

```
stdClass Object
(
    [template] => https://s3-eu-west-1.amazonaws.com/production/templ_1.xml
    [assets] => https://s3-eu-west-1.amazonaws.com/production/assets/223
    [result] => https://s3-eu-west-1.amazonaws.com/production/pdfs/223
)
```

Example 3: Monitoring Queues in Decaf (Java)

We are going to get a bit ahead of ourselves in this section. Operating apps using advanced services like SQS, SNS, and SimpleDB is the subject of Chapter 7. But we wanted to show using SQS in the Java language too. And, as we only consider real examples interesting (the only exception being Hello World, of course), we'll show you excerpts of Java source from Decaf.

If you start using queues as the glue between the components of your applications, you are probably curious about the state. It is not terribly interesting how many messages are in the queue at any given time if your application is working. But if something goes wrong with the pool of readers or writers, you can experience a couple of different situations:

www.it-ebooks.info

- The queue has more messages than normal.
- The queue has too many invisible messages (messages are being processed but not deleted).
- The queue doesn't have enough messages.

We are going to add a simple SQS browser to Decaf. It shows the queues in a region, and you can see the state of a queue by inspecting its attributes. The attributes we are interested in are `ApproximateNumberOfMessages` and `ApproximateNumberOfMessagesNotVisible`. We already have all the mechanics in place to monitor certain aspects of your infrastructure automatically; we just need to add appropriate calls to watch the queues.

Getting the queues

For using the services covered in this chapter, there is AWS SDK for Android. To set up the SQS service, all you need is to pass it your account credentials and set the right endpoint for the region you are working on. In the Decaf examples, we are using the `us-east-1` region. If you are not interested in Android and just want to use plain Java, there is the AWS SDK for Java, which covers all the services. All the examples given in this book run with both sets of libraries.

In this example, we invoke the `ListQueues` action, which returns a list of queue URLs. Any other information about the queues (such as current number of messages, time when it was created, etc.) has to be retrieved in a separate call passing the queue URL, as we show in the next sections.

If you have many queues, it is possible to retrieve just some of them by passing a *queue name prefix* parameter. Only the queues with names starting with that prefix will be returned:

```
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClient;

// ...

// prepare the credentials
String accessKey = "AKIAIGKECZXA7AEIJLMQ";
String secretKey = "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";

// create the SQS service
AmazonSQS sqsService = new AmazonSQSClient(
            new BasicAWSCredentials(accessKey, secretKey));

// set the endpoint for us-east-1 region
sqsService.setEndpoint("https://sqs.us-east-1.amazonaws.com");

// get the current queues for this region
this.queues = sqsService.listQueues().getQueueUrls();
```


www.it-ebooks.info

Reading the queue attributes

The attributes of a queue at the time of writing are:

- The approximate number of visible messages it contains. This number is approximate because of the distributed architecture on which SQS is implemented, but generally it should be very close to reality.
- The approximate number of messages that are not visible. These are messages that have been retrieved by some component to be processed but have not yet been deleted by that component, and the visibility timeout is not over yet.
- The visibility timeout. This is how long a message can be in invisible mode before SQS decides that the component responsible for it has failed and puts the message back in the queue.
- The timestamp when the queue was created.
- The timestamp when the queue was last updated.
- The permissions policy.
- The maximum message size. Messages larger than the maximum will be rejected by SQS.
- The message retention period. This is how long SQS will keep your messages if they are not deleted. The default is four days. After this period is over, the messages are automatically deleted.

You can change the visibility timeout, policy, maximum message size, and message retention period by invoking the `SetQueueAttributes` action.

You can indicate which attributes you want to list, or use `All` to get the whole list, as done here:

```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClient;
import com.amazonaws.services.sqs.model.GetQueueAttributesRequest;

// ...

GetQueueAttributesRequest request = new GetQueueAttributesRequest();

// set the queue URL, which identifies the queue (hardcoded for this example)
String queueURL = "https://queue.amazonaws.com/205414005158/queue4";
request = request.withQueueUrl(queueURL);

// we want all the attributes of the queue
request = request.withAttributeNames("All");

// make the request to the service
this.attributes = sqsService.getQueueAttributes(request).getAttributes();
```

Figure 4-4 shows a screenshot of our Android application, listing attributes of a queue.

www.it-ebooks.info

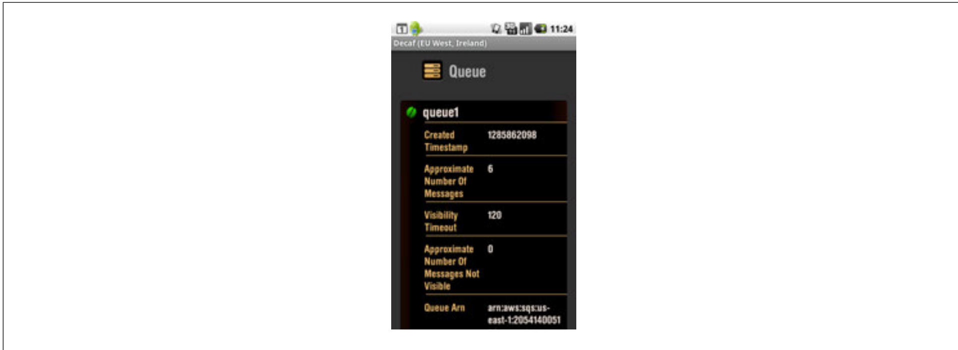


Figure 4-4. Showing the attributes of a queue

Checking a specific queue attribute

If we want to check the number of messages in a queue and trigger an alert (email message, Android notification, SMS, etc.), we can request the attribute `ApproximateNumberOfMessages`:

```
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClient;
import com.amazonaws.services.sqs.model.GetQueueAttributesRequest;

// ...

// get the attribute ApproximateNumberOfMessages for this queue
GetQueueAttributesRequest request = new GetQueueAttributesRequest();
String queueURL = "https://queue.amazonaws.com/205414005158/queue4";
request = request.withQueueUrl(queueURL);
request = request.withAttributeNames("ApproximateNumberOfMessages");

Map<String, String> attrs = sqsService.getQueueAttributes(request).getAttributes();

// get the approximate number of messages in the queue
int messages = Integer.parseInt(attrs.get("ApproximateNumberOfMessages"));

// compare with max, the user's choice for maximum number of messages
if (messages > max) {
    // if number of messages exceeds maximum,
    // notify the user using Android notifications...
    // ...
}
```


www.it-ebooks.info

All the AWS web service calls can throw exceptions—`AmazonServiceException`, `AmazonClientException`, and so on—signaling different errors. For SQS calls you could get errors such as "Invalid attribute" and "Nonexistent queue." In general, you could get other errors common to all web services such as "Authorization failure" or "Invalid request." We are not showing the `try-catch` in these examples, but in your application you should make sure the different error conditions are handled.

SimpleDB

AWS says that SimpleDB is "a highly available, scalable, and flexible nonrelational data store that offloads the work of database administration." There you have it! In other words, you can store an extreme amount of structured information without worrying about security, data loss, and query performance. And you pay only for what you use.

SimpleDB is not a relational database, but to explain what it is, we will compare it to a relational database since that's what we know best. SimpleDB is not a database server, so therefore there is no such thing in SimpleDB as a database. In SimpleDB, you create *domains* to store related *items*. Items are collections of *attributes*, or key-value pairs. The attributes can have multiple values. An item can have 256 attributes and a domain can have one billion attributes; together, this may take up to 10 GB of storage.

You can compare a domain to a table, and an item to a record in that table. A traditional relational database imposes the structure by defining a schema. A SimpleDB domain does not require items to be all of the same structure. It doesn't make sense to have all totally different items in one domain, but you can change the attributes you use over time. As a consequence, you can't define indexes, but they are implicit: every attribute is indexed automatically for you.

Domains are distinct—they are on their own. Joins, which are the most powerful feature in relational databases, are not possible. *You cannot combine the information in two domains with one single query*. Joins were introduced to reconstruct normalized data, where normalizing data means ripping it apart to avoid duplication.

Because of the lack of joins, there are two different approaches to handling relations (previously handled by joins). You can either introduce duplication (e.g., store employees in the employer domain and vice versa), or you can use multiple queries and combine the data at the application level. If you have data duplication and if several applications write to your SimpleDB domains, each of them will have to be aware of this when you make changes or add items to maintain consistency. In the second case, each application that reads your data will need to aggregate information from different domains.

There is one other aspect of SimpleDB that is important to understand. If you add or update an item, it does not have to be immediately available. SimpleDB reserves the

www.it-ebooks.info

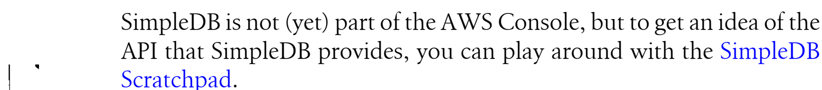
right to take some time to process the operations you fire at it. This is what is called *eventual consistency*, and for many kinds of information, it is not a problem.

But in some cases, you need the latest, most up-to-date information. Think of an online auction website like eBay, where people bid for different items. At the moment a purchase is made, it's important that the right—latest—price is read from the database. To address those situations, SimpleDB introduced two new features in early 2010: *consistent read* and *conditional put/delete*. A consistent read guarantees to return values that reflect all previously successful writes. Conditional put/delete guarantees that a certain operation is performed only when one of the attributes exists or has a particular value. With this, you can implement a counter, for example, or implement locking/concurrency.

We have to stress that SimpleDB is a *service*, and as such, it solves a number of problems for you. Indexing is one we already mentioned. High availability, performance, and infinite scalability are other benefits. You don't have to worry about replicating your data to protect it against hardware failures, and you don't have to think of what hardware you are using if you have more load, or how to handle peaks. Also, the software upgrades are taken care of for you.

But even though SimpleDB makes sure your data is safe and highly available by seamlessly replicating it in several data centers, Amazon itself doesn't provide a way to manually make backups. So if you want to protect your data against your own mistakes and be able to revert back to previous versions, you will have to resort to third-party solutions that can back up SimpleDB data, for example to S3.

Table 4-1 highlights some of the differences between the SimpleDB data store and relational databases.



SimpleDB is not (yet) part of the AWS Console, but to get an idea of the API that SimpleDB provides, you can play around with the SimpleDB Scratchpad.

Table 4-1. SimpleDB data store versus relational databases

Relational databases	SimpleDB
Tables are organized in databases	No databases; all domains are loose in your AWS account
Schemas to define table structure	No predefined structure; variable attributes
Tables, records, and columns	Domains, items, and attributes
Columns have only one value	Attributes can have multiple values
You define indexes manually	All attributes are automatically indexed
Data is normalized; broken down to its smallest parts	Data is not always normalized
Joins are used to denormalize	No joins; either duplicate data or multiple queries
Transactions are used to guarantee data consistency	Eventual consistency, consistent read, or conditional put/delete

www.it-ebooks.info

Use Cases for SimpleDB

So what can you do with SimpleDB? It is different enough from traditional relational databases that you need to approach your problem from other angles, yet it is similar enough to make that extremely difficult.

AWS itself also seems to struggle with this. If you look at the featured use cases, they mention logging, online games, and metadata indexing. Logging is suitable for SimpleDB, but you do have to realize you can't use SimpleDB for aggregate reporting: there are no aggregate functions such as SUM, AVERAGE, MIN, etc. Metadata indexing is a very good pattern of applying SimpleDB to your problem; you can have data stored in S3 and use SimpleDB domains to store pointers to S3 objects with more information about them (metadata). It is very quick and easy to search and query this metadata.

Another class of problems SimpleDB is perfect for is sharing information between isolated components of your application (decoupling). Where we use SQS for communicating actions or messages, SimpleDB provides a way to share indexed information, i.e., information that can be searched. A SimpleDB item is limited in size, but you can use S3 for storing bigger objects, for example images and videos, and point to them from SimpleDB. You could call this metadata indexing.

Other classes of problems where SimpleDB is useful are:

Loosely coupled systems

This is our favorite use of SimpleDB so far. Loosely coupled components share information, but are otherwise independent.

Suppose you have a big system and you suddenly realize you need to scale by separating components. You have to consider what to do with the data that is shared by the resulting components. SimpleDB has the advantage of having no setup or installation overhead, and it's not necessary to define the structure of all your data in advance. If the data you have to share is not very complex, SimpleDB is a good choice. Your data will be highly available for all your components, and you will be able to retrieve it quickly (by selecting or searching) thanks to indexing.

Fat clients

For years, everyone has been working on thin clients; the logic of a web application is located on the server side. With Web 2.0, the clients are getting a bit fatter, but still, the server is king. The explosion of smartphone apps and app stores shows that the fat client is making a comeback. These new, smarter apps do a lot themselves, but in the age of the cloud they can't do everything, of course. SimpleDB is a perfect companion for this type of system: self-contained clients operating on cloud-based information. Not really thick, but certainly not thin: thick-thin-clients.

One advantage of SimpleDB here is that it's ready to use right away. There is no setup or administration hassle, the data is secure, and, most importantly, SimpleDB provides access through web services that can be called easily from these clients.

www.it-ebooks.info

Typically, many applications take advantage of storing data in the cloud to build different kinds of clients—web, smartphone, desktop—accessing the same data.

Large (linear) datasets

The obvious example of a large, simple dataset in the context of Amazon is books. A quick calculation shows that you can store 31,250,000 books in one domain, if 32 attributes is enough for one book. We couldn't find the total number of different titles on Amazon.com, but Kindle was reported to offer 500,000 titles in April 2010. Store PDFs in S3, and you are seriously on your way to building a bookstore.

Other examples are music and film—much of the online movie database IMDb's functionality could be implemented with SimpleDB. If you need to scale and handle variations in your load, SimpleDB makes sure you have enough resources, and guarantees good performance when searching through your data.

Hyperpersonal

A new application of cloud-based structured storage (SimpleDB) can be called hyperpersonal storage. For example, saving the settings or preferences of your applications/devices or your personal ToDo list makes recovery or reinstallation very easy. Everyone has literally hundreds (perhaps even thousands) of these implicit or explicit little lists of personal information.

This is information that is normally not shared with others. It is very suitable for SimpleDB because there is no complex structure behind it.

Scale

Not all applications are created equal, and "some are more equal than others." Most public web applications would like to have as many users as Amazon.com, but the reality is different. If you are facing the problem of too much load, you have probably already optimized your systems fully. It is time to take more dramatic measures.

As described in Chapter 1, the most drastic thing you can do is eliminate joins. If you get rid of joins, you release part of your information; it becomes isolated and can be used independently of other information. You can then move this free information to SimpleDB, which takes over responsibilities such as scalability and availability while giving you good performance.

Example 1: Storing Users for Kulitzer (Ruby)

On the Web you can buy, sell, and steal, but you can't usually organize a contest. That is why Kulitzer is one of a kind. Contests are everywhere. There are incredibly prestigious contests like the Cannes Film Festival or the Academy Awards. There are similar prizes in every country and every school. There are other kinds of contests like *American Idol*, and every year thousands of girls want to be the prom queen.

www.it-ebooks.info

Kulitzer aims to be one platform for all. If you want to host a contest, you are more than welcome to. But some of these festivals require something special—they want their own Kulitzer. At this moment, we want to meet this demand, but we do not want to invest in developing something like Google Sites where you can just create your own customized contest environment. We will simply set up a separate Kulitzer environment, designed to the wishes of the client.

But the database of users is very important, and we want to keep it on our platform. We believe that, in the end, Kulitzer will be the brand that gives credibility to the contests it hosts. Kulitzer is the reputation system for everyone participating in contests. This means that our main platform and the specials are all built on one user base, with one reputation system, and a recognizable experience for the users (preferences are personal, for example; you take them with you). We are going to move the users to SimpleDB so they can be accessed both by the central Kulitzer and by the separate special Kulitzer environments. Figure 4-5 shows the new architecture. The new components are the SimpleDB users domain and the new EC2 instances for separate contests.

> Figure 4-5 does not show the RDS instance used for the database, which we introduced in Chapter 1. With this change, each new special Kulitzer would use an RDS instance of its own for its relational data. In fact, each Kulitzer environment would be run on a different AWS account. To simplify the figure, ELBs are not shown.

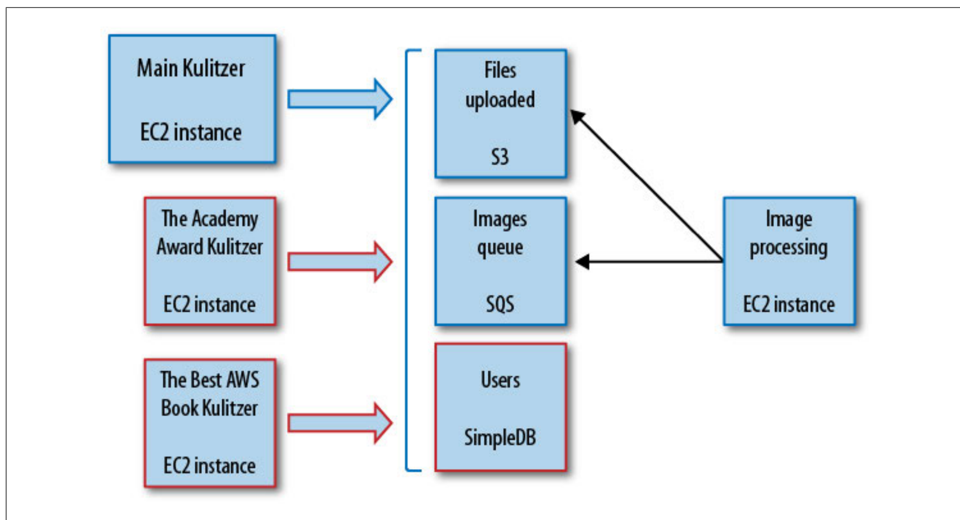


Figure 4-5. Introducing SimpleDB for sharing user accounts

www.it-ebooks.info

The solution of moving the users to SimpleDB falls into several of the classes of problems described above. It is a large linear dataset we need to scale, and we also deal with disparate (decoupled) systems.

In the examples below, we show how to create an item in a `users` domain, and then how to retrieve a user item using Ruby.

Adding a user

To add a user, all you need is something like the following. We use the RightAWS Ruby library. The attribute `id` will be the item name, which is always unique in SimpleDB. When using SimpleDB with RightAWS, we need an additional gem called `uuidtools`, which you can install with `gem install uuidtools`:

```
require 'rubygems'
require 'right_aws'
require 'sdb/active_sdb'

RightAws::ActiveSdb.establish_connection("AKIAIGKECZXA7AEIJLMQ",
    "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")

# right_aws' simpledb is still alpha. it works, but it feels a bit
# experimental and does not resemble working with SQS. Anyway, you need
# to subclass Base to access a domain. The name of the class will be
# lowercased to domain name.
class Users < RightAws::ActiveSdb::Base
end

# create domain users if it doesn't exist yet,
# same as RightAws::ActiveSdb.create_domain("users")
Users.create_domain

# add Arjan
# note: id is used by right_aws sdb as the name, and names are unique
# in simpledb. but create is sort of idempotent, doens't matter if you
# create it several times, it starts to act as an update.
Users.create(
            'id' => 'arjanvw@gmail.com',
            'login' => 'mushimushi',
            'description' =>
                'total calligraphy enthusiast ink in my veins!!',
            'created_at' => '1241263900',
            'updated_at' => '1283845902',
            'admin' => '1',
            'password' =>
                '33a1c623d4f95b793c2d0fe0cadc34e14f27a664230061135',
            'salt' => 'Koma659Z3Zl8zXmyyyLQ',
            'login_count' => '22',
            'failed_login_count' => '0',
            'last_request_at' => '1271243580',
            'active' => '1',
            'agreed_terms' => '1')
```


www.it-ebooks.info

Getting a user

Once you know who you want, it is easy and fast to get everything. This particular part of the RightScale SimpleDB Ruby implementation does not follow the SQS way. SQS is more object oriented and easier to read. But the example below is all it takes to get a particular item. If you want to perform more complex queries, it doesn't get much more difficult. For example, getting all admins can be expressed with something like `administrators = Users.find( :all, :conditions=> [ "['admin'=?]", "1"])`:

```
require 'rubygems'
require 'right_aws'
require 'sdb/active_sdb'

RightAws::ActiveSdb.establish_connection("AKIAIGKECZXA7AEIJLMQ",
    "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")

class Users < RightAws::ActiveSdb::Base
end

# get Arjan
arjan = Users.find('arjanvw@gmail.com')
# reload to get all the attributes
arjan.reload

puts arjan.inspect
```

> As we were writing this book, we found that there are not many alternatives to a good SimpleDB solution for Ruby. We chose to use Right-AWS because of consistency with the SQS examples, and because we found it was the most comprehensive of the libraries we saw. The implementation of SimpleDB is a bit "Railsy": it generates IDs automatically and uses those as the names for the items. You can overwrite that behavior, as we did, at your convenience.

Example 2: Sharing Marvia Accounts and Templates (PHP)

With Marvia, we started to implement some of our ideas regarding quality of service, a finer-grained approach to product differentiation ("Example 2: Priority PDF Processing for Marvia (PHP)" on page 77). We created a priority lane for important jobs using SQS, giving the user the option of preferential treatment. We introduced flexibility at the job level, but we want to give more room for product innovation.

The current service is an all-you-can-eat plan for creating your PDFs. This is interesting, but for some users it is too much. In some periods, they generate tens or even hundreds of PDFs per week, but in other periods they only require the service occasionally. These users are more than willing to spend extra money during their busy weeks if they can lower their recurring costs.

www.it-ebooks.info

We want to introduce this functionality gradually into our application, starting by implementing a fair-use policy on our entry-level products. For this, we need two things: we need to keep track of actual jobs (number of PDFs created), and we need a phone to inform customers if they are over the limit. If we know users consistently breach the fair use, it is an opportunity to upgrade. So for now we only need to start counting jobs.

At the same time, we are working toward a full-blown API that implements many of the features of the current application and that will be open for third parties to use. Extracting the activity per account information from the application and moving it to a place where it can be shared is a good first step in implementing the API. So the idea is to move accounts to SimpleDB. SimpleDB is easily accessible by different components of the application, and we don't have to worry about scalability. Figure 4-6 shows the new Marvia ecosystem.

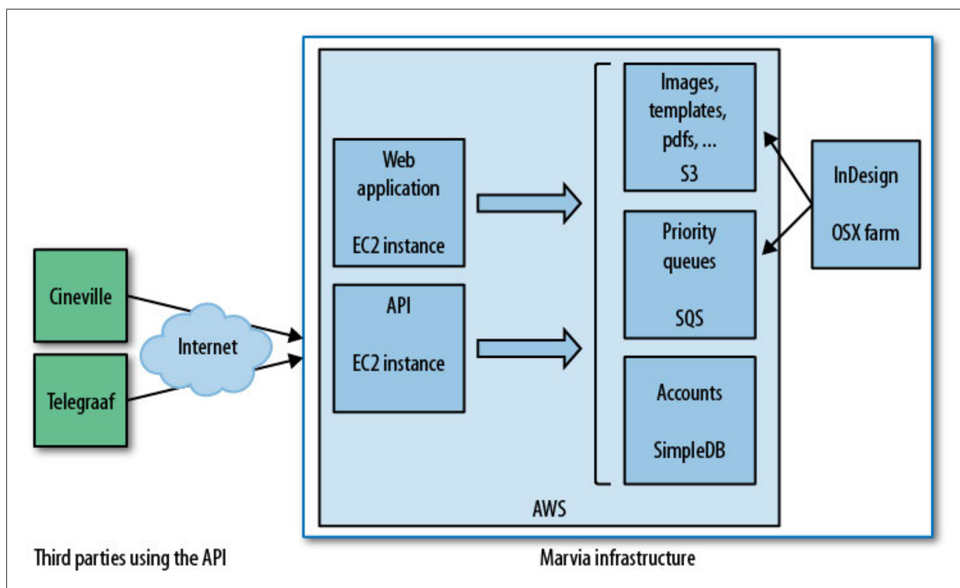


Figure 4-6. Using SimpleDB in Marvia for sharing user accounts

Adding an account

Adding an item to a domain in SimpleDB is just as easy as sending a message to a queue in SQS. An item always has a `name`, which is unique within its domain; in this case, we chose the ID of the account as the name. We are going to count the number of PDFs generated for all accounts, but we do want to share if an account is restricted under the fair-use policy of Marvia.

As with the SQS `create_queue` method, the `create_domain` method is idempotent (it doesn't matter if you run it twice; nothing changes). We need to set the region because we want the lowest latency possible and the rest of our infrastructure is in Europe. And

www.it-ebooks.info

instead of having to use JSON to add structure to the message body, we can add multiple attributes as name/value pairs. We'll have the attribute `fair`, indicating whether this user account is under the fair-use policy, and `PDFs`, containing the number of jobs done. We want both attributes to have a singular value, so we have to specify that we don't want to add, but rather replace, existing attribute/value pairs in the `put_attributes` method:

```
<?php
    require_once('/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');
    define('AWS_ACCOUNT_ID', '457964863276');

    # construct the message (use zero padding to handle
    # simpledb's lexicographic ordering)
    $account = array(
        'fair' => 'yes',
        'PDFs' => sprintf('%05d', '0'));

    $sdb = new AmazonSDB();
    $sdb->set_region($sdb::REGION_EU_W1);

    $accounts = $sdb->create_domain('accounts');
    $accounts->isOK() or
            die('could not create domain accounts');

    $response = $sdb->put_attributes('accounts',
            'jurg@9apps.net', $account, true);

    pr($response->body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```

Getting an account

If you want to get all attributes of a specific item, you can use the `get_attributes` method. Just state the domain name and item name, and you will get a `CFSimpleXML` object with all attributes. This is supposed to be easy, but we don't find it very straightforward. But if you understand it, you can parse XML with very brief statements, which is convenient compared to other ways of dealing with XML:

```
<?php
    require_once('/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');
    define('AWS_ACCOUNT_ID', '457964863276');

    $sdb = new AmazonSDB();
    $sdb->set_region($sdb::REGION_EU_W1);
```


www.it-ebooks.info

```
    $accounts = $sdb->create_domain('accounts');
    $accounts->isOK() or
            die('could not create domain accounts');

    $response = $sdb->get_attributes('accounts',
            'jurg@9apps.net');

    pr($response->body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```

As output when invoking this script, we will see something like this:

```
CFSimpleXML Object
(
    [@attributes] => Array
        (
            [ns] => http://sdb.amazonaws.com/doc/2009-04-15/
        )

    [GetAttributesResult] => CFSimpleXML Object
        (
            [Attribute] => Array
                (
                    [0] => CFSimpleXML Object
                        (
                            [Name] => PDFs
                            [Value] => 00000
                        )

                    [1] => CFSimpleXML Object
                        (
                            [Name] => fair
                            [Value] => yes
                        )

                )

        )

    [ResponseMetadata] => CFSimpleXML Object
        (
            [RequestId] => eec8b61d-4107-0f1c-45b2-219f5f0b895a
            [BoxUsage] => 0.0000093282
        )

)
```

Incrementing the counter

Before the introduction of conditional puts and consistent reads, some things were impossible to guarantee: for example, atomically incrementing the value of an attribute. But with conditional puts, we can do just that. SimpleDB does not have an increment operator, so we have to first get the value of the attribute we want to increment. It is

www.it-ebooks.info

possible that someone else updated that particular attribute. We can try, but the conditional put will fail. Not a problem—we can just try again. (We just introduced a possible race condition, but we are not generating thousands of PDFs a second; if we do 100 a day per account, it is a lot.)

To be sure we have the most up-to-date value for this attribute, we do a consistent read. You can consider a consistent read as a flush for writes (puts)—using consistent reads forces all operations to be propagated over the replicated instances of your data. A regular read can get information from a replicated instance that does not yet have the latest updates in the system. A consistent read can be slower, especially if there are operations to be forced to propagate:

```
<?php
    require_once('/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');
    define('AWS_ACCOUNT_ID', '457964863276');

    $sdb = new AmazonSDB();
    # $sdb->set_region($sdb::REGION_EU_W1);

    $accounts = $sdb->create_domain( 'accounts');
    $accounts->isOK() or
            die('could not create domain accounts');

    # we have to be a bit persistent; even though it
    # is unlikely someone else might have incremented
    # during our operation
    do {
        $response = $sdb->get_attributes('accounts',
            'jurg@9apps.net', 'PDFs', array( 'ConsistentRead' => 'true'));
        $PDFs = (int)$response->body->Value(0);

        $account = array('PDFs' => sprintf( '%05d', $PDFs + 1));
        $response = $sdb->put_attributes(
            'accounts',
            'jurg@9apps.net',
            $account,
            true,
            array(
            'Expected.Name' => 'PDFs',
            'Expected.Value' => sprintf( '%05d', $PDFs)));
    } while($response->isOK() === FALSE);
?>
```

Example 3: SimpleDB in Decaf (Java)

Decaf is a mobile tool, and it offers a unique way of interacting with your AWS applications. If your app is on AWS and you use SimpleDB, there are a couple of interesting things you can learn on the go.

www.it-ebooks.info

Sometimes you're not at the office, but you would like to search or browse your items in SimpleDB. Perhaps you're at a party and you want to know if someone you meet is a Kulitzer user. Or perhaps you run into a customer at a networking event and you want to know how many PDFs have been generated by her.

Another example addresses the competitive thrill seeker in us. If we have accounts in SimpleDB, we might want to monitor the size of that domain. We want to know when we hit 1,000 and, after that, 10,000.

So that is what we are going to do here. We are going to implement a basic SimpleDB browser that can search for and monitor certain attributes.

> If you use the Eclipse development environment for Java, you can try out the AWS Toolkit for Eclipse, which provides some nice graphical tools for managing SimpleDB. You can manage your domains and items, and make select queries in a Scratchpad.

Listing domains

Listing existing domains is simple. The `ListDomains` action returns a list of domain names. This list has, by default, a maximum of 100 elements (you can change that by adding a parameter in the request), and if there are more than the maximum, you get a token to get the next "page" of results:

```
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.AmazonSimpleDBClient;
import com.amazonaws.services.simpledb.model.ListDomainsRequest;
import com.amazonaws.services.simpledb.model.ListDomainsResult;

// ...

// prepare the credentials
String accessKey = "AKIAIGKECZXA7AEIJLMQ";
String secretKey = "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";

// create the SimpleDB service
AmazonSimpleDB sdbService = new AmazonSimpleDBClient(
    new BasicAWSCredentials(accessKey, secretKey));


// set the endpoint for us-east-1 region
sdbService.setEndpoint("https://sdb.amazonaws.com");

String nextToken = null;
ListDomainsRequest request = new ListDomainsRequest();

List<String> domains = new ArrayList<String>();

// get the existing domains for this region
do {
```


www.it-ebooks.info

```
        if (nextToken != null) request = request.withNextToken(nextToken);
        ListDomainsResult result = sdbService.listDomains(request);
        nextToken = result.getNextToken();
        domains.addAll(result.getDomainNames());
    } while (nextToken != null);

    // show the domains in a list...
```

Listing items in a domain: select

The way to list the items in a domain is to execute a `select`. The syntax is similar to that of an SQL `select` statement, but of course there are no joins; it is only for fetching items from one domain at a time. The results are paginated, as in the case of domains, in a way that each response never exceeds 1 MB.

The following retrieves all the items of a given domain:

```
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.AmazonSimpleDBClient;
import com.amazonaws.services.simpledb.model.Item;
import com.amazonaws.services.simpledb.model.SelectRequest;
import com.amazonaws.services.simpledb.model.SelectResult;

// ...

// determine which domain we want to list
String domainName = ...

AmazonSimpleDB simpleDBService = ...;

// initialize list of items
List<Item> items = new ArrayList<Item>();

// nextToken == null is the first page
String nextToken = null;

// set the select expression which retrieves all the items from this domain
SelectRequest request = new SelectRequest("select * from " + domainName);

do {
    if (nextToken != null) request = request.withNextToken(nextToken);
    // make the request to the service
    SelectResult result = simpleDBService.select(request);

    nextToken = result.getNextToken();
    items.addAll(result.getItems());
} while (nextToken != null);


// show the items...
```


www.it-ebooks.info

In these examples, we retrieve all the pages using NextToken and then we show them all to simplify the example. When the number of items is very big and you are showing them to a user, it's probably best to get a page, show it, then get more pages (if necessary). Otherwise, the user might have to wait a long time to see something. Also, the amount of data might be too big to keep in memory.

Of course, with a `select` expression, we can do more complex filtering by adding a `where` clause, listing only certain attributes and sorting the items by one of the attributes. For example, if we have the `users` table and we are looking for a user whose name is "Juan" and whose surname starts with "P", we would do something like this:

```
select name, surname from users
    where name like 'Juan%' intersection surname like 'P%'
```

In addition to `'*'` and a list of attributes, the output of a `select` can also be `count(*)` or `itemName()` to return the name of the item.

Getting domain metadata

If you'd like to know the total number of items in your domain, one way is to use a `select`:

```
select count(*) from users
```

Another option is to retrieve the metadata of the domain, like we do here:

```
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.AmazonSimpleDBClient;
import com.amazonaws.services.simpledb.model.DomainMetadataRequest;
import com.amazonaws.services.simpledb.model.DomainMetadataResult;

// determine which domain we want to list
String domainName = ...

AmazonSimpleDB simpleDBService = ...;

// prepare the DomainMetadata request for this domain
DomainMetadataRequest request = new DomainMetadataRequest(domainName);

DomainMetadataResult result = simpleDBService.domainMetadata(request);

// we are interested in the total amount of items
long totalItems = result.getItemCount();

// show results
System.out.println("Domain metadata: " + result);
System.out.println("The domain " + domainName + " has " +
    totalItems + " items.");
```


www.it-ebooks.info

The `DomainMetadataResult` can look similar to this:

```
{ItemCount: 210,
 ItemNamesSizeBytes: 3458,
 AttributeNameCount: 2,
 AttributeNamesSizeBytes: 18,
 AttributeValueCount: 245,
 AttributeValuesSizeBytes: 7439,
 Timestamp: 1287263703, }
```

SNS

Both SQS and SimpleDB are kind of passive, or *static*. You can add things, and if you need something from it, you have to pull. This is OK for many services, but sometimes you need something more disruptive—you need to push instead of pull. This is what Amazon SNS gives us. You can push information to any component that is listening, and the messages are delivered right away. Table 4-2 shows a comparison of SNS and SQS.

Table 4-2. Comparing SNS to SQS

SQS	SNS
Doesn't require receivers to subscribe	Receivers subscribe and confirm
Each message is normally handled by one receiver, then deleted	Notifications are "broadcasted" to all subscribers
Pull model: messages are read by receivers when they want	*Push* model: notifications are automatically pushed to all subscribers

SNS is not an easy service, but it is incredibly versatile. Luckily, we are all living in "the network society," so the essence should be familiar to most of us. It is basically the same concept as a mailing list or LinkedIn group—there is something you are interested in (a `topic`, in SNS-speak), and you show that interest by subscribing. Once the topic verifies that you exist by confirming the subscription, you become part of the group receiving messages on that topic.

Of course, notifications are not new—we have been using them for a while now. In software systems, one of the more successful models of publish/subscribe communication is the *event model*. Events are part of most high-level programming languages like Java and Objective-C. In a software system that spans multiple components, there is the notion of an event. In the field of web development, events are abundant, though mostly implicit. Every PHP script or Rails Action Controller method (public, of course) is an event handler; it reacts to a user requesting a page or another piece of software talking to an API.

www.it-ebooks.info

There is one important difference between SNS and the events in programming languages as we know them: permissions are implemented differently. In Java, for example, you don't need explicit permissions to subscribe to events from a certain component. This is left to the environment the application operates in. In the Android mobile platform, on the other hand, you have to request access to, say, the compass or the Internet in your application's manifest file. If you don't specify that you need those privileges, you get a compile-time error. Because SNS is much more public, not just confined to one user on one machine, permissions are part of the events themselves. You need permission from the topic owner to receive information, and you need permission from the recipient to send information.

SNS can be seen as an event system, but how does it work? In SNS, you create topics. Topics are the conduits for sending (publishing) and receiving *messages*, or events. Anyone with an AWS account can subscribe to a topic, though that doesn't mean they will be automatically permitted to receive messages. And the topic owner can subscribe non-AWS users on their behalf. Every subscriber has to explicitly opt in, though that term is usually related to mailing lists and spam. But it is the logical consequence in an open system like the Web (you can see this as the equivalent of border control in a country).

The most interesting thing about SNS has to do with the subscriber, the recipient of the messages. A subscriber can configure an *endpoint*, specifying how and where the message will be delivered. Currently SNS supports three types of endpoints: HTTP/HTTPS, email, and SQS. And this is exactly the reason we feel it is more than a notification system. You can integrate an API using SNS, enabling totally different ways of execution.

But, even as just a notification system, SNS rocks. (And Amazon promises to support SMS endpoints, as well.)

> Happily, SNS is now included in the AWS Console, so you can take a look for yourself and experiment with topics, subscriptions, and messages.

Example 1: Implementing Contest Rules for Kulitzer (Ruby)

A contest is the central component of Kulitzer. Suppose our company, 9Apps, needs a new identity. With our (still) limited budget, we can't ask five high-profile marketing agencies to come to us and pitch for this project—we just don't have that much money. What we can do, however, is ask a number of designers in our community to participate in a contest organized on Kulitzer. And because we use Kulitzer, we can easily invite our partners and customers to vote on the best design.

In our quest for flexibility, we realize that this problem would benefit from using SNS. The lifecycle of a contest is littered with events. We start a contest, we invite participants

www.it-ebooks.info

and a jury, and admission is opened and closed, as is the voting. And during the different periods of the contest, we want to notify different interested parties of progress. Later, we can add all sorts of components that "listen to a contest" and take action when certain events occur, for example, leaving messages on Facebook walls or sending tweets.

We could implement this with SNS by having a few central topics, like "registration" for indicating that a contest is open for participants, "admission" when the admission process starts, and "judging" when the voting starts. We could then have subscribers to each of these topics, but they would receive notifications related to all existing contests. It would move the burden of implementing security to the components themselves.

Instead, we choose to implement this differently. The idea is that each contest has its own topics for the kind of events we mentioned. The contest will handle granting/revoking permissions with SNS so that listeners are simpler to implement. It will give us much more flexibility to add features to Kulitzer later on.

The current limits for SNS —which is in beta— state that you can have a maximum of 100 topics per account. This would clearly limit us when choosing to have several topics per contest. But we contacted Amazon about this issue, and its reply expressed interest in our use case, so it will not be a problem.

This is not the first time we needed to surpass the limits imposed by AWS. When this happens, Amazon requests that you explain your case. Amazon will then analyze it and decide if these limitations can be lifted for you.

Preparing tools for SNS in Ruby

The RightAWS library doesn't officially support SNS yet, but we found a branch by Brendon Murphy that does. The branch is called `add_sns`, and you can download it from GitHub and create the gem yourself.

You will need to install some dependencies:

```
sudo gem install hoe
sudo gem install rcov
```

Then download the sources, extract them, and run the following to generate the gem in the *pkg* directory:

```
rake gem
```

To install it, run the following from the *pkg* directory:

```
sudo gem install right_aws-1.11.0.gem
```


www.it-ebooks.info

If you installed the official version of RightAWS before, you can uninstall it before (or after) installing this one, to make sure you use the right one. To do that, execute the following:

```
sudo gem uninstall right_aws
```

Topics per contest

Here, we show how to create the topics for a specific contest:

```
require "right_aws"

sns = RightAws::Sns.new("AKIAIGKECZXA7AEIJLMQ",
                        "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")
contest_id = 12

# create the topics

registration = sns.create_topic("#{contest_id}_registration")
# set a human readable name for the topic
registration.display_name = "Registration process begins"

admission = sns.create_topic("#{contest_id}_admission")
admission.display_name = "Admission process begins"

judging = sns.create_topic("#{contest_id}_judging")
judging.display_name = "Judging process begins"
```

Subscribing to registration updates

There are two steps to subscribing to a topic:

1. Call the `subscribe` method, providing the type of endpoint (`email`, `email-json`, `http`, `https`, `sqs`).
2. Confirm the subscription by calling `confirm_subscription`.

For the second step, SNS will send the endpoint a request for confirmation in the form of an email message or an HTTP request, depending on the type of endpoint. In all cases, a string `token` is provided, which needs to be sent back in the `confirm_subscription` call. If the endpoint is email, the user just clicks on a link to confirm the subscription, and the token is sent as a query string parameter of the URL (shown in Figure 4-7).

If the endpoint is `http`, `https`, or `email-json`, the token is sent as part of a JSON object.

Step one is subscribing some email addresses to a topic:

```
require "right_aws"

sns = RightAws::Sns.new("AKIAIGKECZXA7AEIJLMQ",
                        "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")
contest_id = 12

# get the judging topic
```


www.it-ebooks.info

```
topic = sns.create_topic("#{contest_id}_judging")

# Subscribe a few email addresses to the topic.
# This will return "Pending Confirmation" and send a confirmation
# email to the target email address which contains a "Confirm Subscription"
# link back to Amazon Web Services.  Other possible subscription protocols
# include http, https, email, email-json and sqs

%w[alice@example.com bob@example.com].each do |email|
  topic.subscribe("email", email)
end

# Let's subscribe this address as email-json, imagining it
# is an email box feeding to a script parsing the content.  Note
# that the confirmation email this address receives will contain
# content as JSON
topic.subscribe("email-json", "carol-email-bot@example.com")

# Another kind of endpoint is http
topic.subscribe("http",
    "http://www.kulitzer.com/contests/update/#{contest_id}_judging")
```

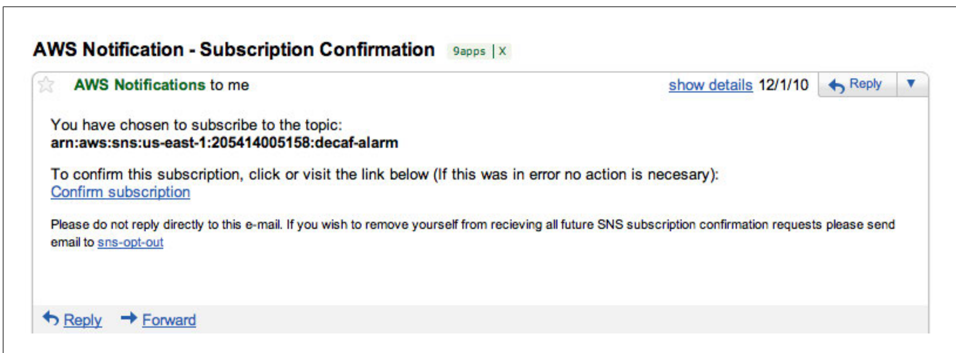


Figure 4-7. Subscription confirmation via email

In step two, the endpoint confirms the subscription by sending back the token:

```
require "right_aws"

# Now, imagine you subscribe an internal resource to the topic and instruct
# it to use the http protocol and post to a url you provide.  It will receive
# a post including the following bit of JSON data:
=begin
{
 "Type" : "SubscriptionConfirmation",
 "MessageId" : "b00e1384-6a4f-4bc5-abd5-9b7f82e3cff4",
 "Token" : "51b2ff3edb4487553c7dd2f29566c2aecada20b9...",
 "TopicArn" : "arn:aws:sns:us-east-1:235698110812:12_judging",
 "Message" : "You have chosen to subscribe to the topic
    arn:aws:sns:us-east-1:235698110812:12_judging.\n
    To confirm the subscription, visit the SubscribeURL included in this
    message.",
```


www.it-ebooks.info

```
 "SubscribeURL" :
    "https://sns.us-east-1.amazonaws.com/?Action=ConfirmSubscription&
    TopicArn=arn:aws:sns:us-east-1:235698110812:12_judging&
        Token=51b2ff3edb4487553c7dd2f29566c2aecada20b9…",
 "Timestamp" : "2011-01-07T11:41:02.417Z",
 "SignatureVersion" : "1",
 "Signature" : "UHWoZfMkpH/FrhICs6AnOcTtjjcj5nBEweVbWgrARD5B..."
}
=end

# You can confirm the subscription with a call to confirm_subscription.
# Note that in order to receive messages published to a topic, the subscriber
# must be confirmed.

sns = RightAws::Sns.new("AKIAIGKECZXA7AEIJLMQ",
                        "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")

arn = "arn:aws:sns:us-east-1:724187402011:12_judging"
topic = sns.topic(arn)

token = "51b2ff3edb4487553c7dd2f29566c2aecada20b9..."
topic.confirm_subscription(token)
```

Publishing messages to a topic

Publish messages to a topic as follows:

```
# Sending a message allows for an optional subject to be included along with
# the required message itself.

# Find the sns topic using the arn.  Note that you can call
# RightAws::SNS#create_topic if you only know the topic name
# and the current AWS api will fetch it for you.  Otherwise,
# you need to store the arn for future lookups.  It's probably
# advisable to store the arn's anyway as they serve as a reliable
# uid.

sns = RightAws::Sns.new("AKIAIGKECZXA7AEIJLMQ",
                        "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn")

arn = "arn:aws:sns:us-east-1:235698110812:12_judging"
topic = sns.topic(arn)

message = "Dear Judges, admission is closed, you can now start evaluating
   participants."
optional_subject = "Judging begins"
topic.send_message(message, optional_subject)
```

Deleting a topic

Of course, when a topic is no longer valid, you can delete it using the following:

```
topic.delete
```


www.it-ebooks.info

Publishing to Facebook

The use of SNS is not limited to sending messages to subscribed users, of course. In Kulitzer, when the status of a contest changes, we might want to update a particular Facebook account to tell the world about it. In Rails, we can implement this by having a separate FacebookController that is subscribed to the different topics and publishes messages to Facebook when notified by SNS of an event.

Example 2: PDF Processing Status (Monitoring) for Marvia (PHP)

The core of Marvia is the PDF engine, for lack of a better term. The PDF engine is becoming a standalone service. We are slowly extracting features from the app, moving them to the PDF engine. We introduced priority queues to distinguish between important and less important jobs. We extracted accounts so we can share information and act on that information, for example keeping track of the number of PDFs generated.

One of the missing parts of the PDF engine is notification. We send jobs to the engine, and it helps us keep track of what is created. But we do not know the status of a job. The only way to learn the status is to check whether the PDF has been generated. It would be great if the process that creates the job could register an event handler to be called with status updates. It could be something simple like a message, or something more elaborate like a service that moves the PDF to a Dropbox account. We could even tell the PDF engine to add a message to a particular queue for printing, for example.

But we'll start with what we need right now: notifying the customer with an email message when the PDF is ready. The emails SNS sends are not really customizable, and we don't want to expose our end users to notification messages with opt-out pointing to Amazon AWS. We want to hide the implementation as much as possible. Therefore, we have notifications sent to our app, which sends the actual email message.

Creating a topic for an account

Creating a topic is not much different from creating a queue in SQS or a domain in SimpleDB. Topic names must be made up of only uppercase and lowercase ASCII letters, numbers, and hyphens, and must be between 1 and 256 characters long. We want a simple mechanism for creating unique topic names, but we still want to be flexible. An easy way is to create a hash of what we would have used if we could. We will create a topic for each Marvia customer account, so we will use the customer's provided email address. In the following, we create a topic for the customer 9apps:

```
<?php
    require_once('/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');

    $sns = new AmazonSNS();
```


www.it-ebooks.info

```
    $sns->set_region($sns::REGION_EU_W1);

    $topic = md5('jurg@9apps.net/status');

    # if topic already exists, create_topic returns it
    $response = $sns->create_topic($topic);
    $response->isOK() or
        die('could not create topic ' . $topic);

    pr($response->body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```

Subscription and confirmation

The endpoint for notification is `http`; we are not sending any sensitive information, so HTTPS is not necessary. But we do want to prepare a nicely formatted email message, so we won't use the standard email endpoint. We can then generate the content of the email message ourselves.

Below, we call `subscribe` to add an `http` endpoint to the topic:

```
<?php
    require_once( '/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');

    $sns = new AmazonSNS();
    $sns->set_region($sns::REGION_EU_W1);

    $topic = md5('jurg@9apps.net/status');

    $response = $sns->create_topic( $topic);
    $response->isOK() or
        die( 'could not create topic ' . $topic);

    # subscribe to the topic by
    # passing topic arn (topic identifier),
    # type of endpoint 'http',
    # and the given url as endpoint
    $response = $sns->subscribe(
        (string)$response->body->TopicArn(0),
        'http',
        'http://ec2-184-72-67-235.compute-1.amazonaws.com/sns/receive.php'
    );

    pr($response->body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```


www.it-ebooks.info

Before messages are delivered to this endpoint, it has to be confirmed by calling `confirm_subscription`. In the code below, you can see that the endpoint is responsible for both confirming the subscription and handling messages. A JSON object is posted, and this indicates if the HTTP POST is a notification or a confirmation. The rest of the information, such as the message (if it is a notification), topic identifier (ARN), and token (if it is a subscription confirmation), is also part of the JSON object:

```
<?php
    require_once( '/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');

    $sns = new AmazonSNS();
    $sns->set_region($sns::REGION_EU_W1);

    # we only accept a POST, before we get the raw input (as json)
    if ( $_SERVER['REQUEST_METHOD'] === 'POST' ) {
        $json = trim(file_get_contents('php://input'));

        $notification = json_decode($json);
        # do we have a message? or do we need to confirm?
        if( $notification->{'Type'} === 'Notification') {

            # a message is text, in our case json so we decode and proceed...

            $message = json_decode($notification->{'Message'});

            # and now we can act on the message
            # ...

        } else if($notification->{'Type'} === 'SubscriptionConfirmation') {

            # we received a request to confirm subscription, let's confirm...

            $response = $sns->confirm_subscription(
                $notification->{'TopicArn'},
                $notification->{'Token'});

            $response->isOK() or
                die("could not confirm subscription for topic 'status'");
        }
    }
?>
```

Publishing and receiving status updates

Publishing a message is not difficult. If you are the owner, you can send a text message to the topic. We need to pass something with a bit more structure, so we use JSON to deliver an object:

www.it-ebooks.info

```php
<?php
    require_once( '/usr/share/php/AWSSDKforPHP/sdk.class.php');

    define('AWS_KEY', 'AKIAIGKECZXA7AEIJLMQ');
    define('AWS_SECRET_KEY', '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');

    $sns = new AmazonSNS();
    $sns->set_region($sns::REGION_EU_W1);

    $topic = md5( 'jurg@9apps.net/status');
    $job = "457964863276";

    $response = $sns->create_topic( $topic);
    $response->isOK() or
        die( 'could not create topic ' . $topic);

    $response = $sns->publish(
        (string)$response->body->TopicArn(0),
        json_encode( array( "job" => $job,
                        "result" => "200",
                        "message" => "Job $job finished"))
    );

    pr( $response->body);

    function pr($var) { print '<pre>'; print_r($var); print '</pre>'; }
?>
```

Example 3: SNS in Decaf (Java)

An interesting idea is to use the phone as a beacon, sending notifications to the world if an unwanted situation occurs. We could do that easily with Decaf on Android. We already monitor our EC2 instances with Decaf, and we could use SNS to notify others. Though interesting, the only useful application of this is to check the health of an app in a much more distributed way. Central monitoring services can only implement a limited number of network origins. In the case of Layar, for example, we can add this behavior to the mobile clients.

For now, we'll implement something else. Just as with SimpleDB and SQS, we are curious about the state of SNS in our application. For queues, we showed how to monitor the number of items in the queue, and for SimpleDB, we did something similar. With SNS, we want to know if the application is operational.

To determine if a particular topic is still working, we only have to listen in for a while by subscribing to the topic. If we see messages, the SNS part of the app works. What we need for this is simple browsing/searching for topics, subscribing to topics with an email address, and sending simple messages.

www.it-ebooks.info

Listing topics

Getting a list of available topics is very simple. The number of results are limited and paginated as in SimpleDB. The code below lists all topics in the given account for the region `us-east-1`:

```
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.sns.AmazonSNS;
import com.amazonaws.services.sns.AmazonSNSClient;
import com.amazonaws.services.sns.model.ListTopicsRequest;
import com.amazonaws.services.sns.model.ListTopicsResult;
import com.amazonaws.services.sns.model.Topic;

// ...

// prepare the credentials
String accessKey = "AKIAIGKECZXA7AEIJLMQ";
String secretKey = "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";

// create the SNS service
AmazonSNS snsService = new AmazonSNSClient(
    new BasicAWSCredentials(accessKey, secretKey));

// set the endpoint for us-east-1 region
snsService.setEndpoint("https://sns.us-east-1.amazonaws.com");

List<Topic> topics = new ArrayList<Topic>();
String nextToken = null;

do {

    // create the request, with nextToken if not empty
    ListTopicsRequest request = new ListTopicsRequest();
    if (nextToken != null) request = request.withNextToken(nextToken);

        // call the web service
        ListTopicsResult result = snsService.listTopics(request);

        nextToken = result.getNextToken();

        // get that list of topics
        topics.addAll(result.getTopics());

// go on if there are more elements
} while (nextToken != null);


// show the list of topics...
```


www.it-ebooks.info

Subscribing to a topic via email

Given the list of topics, we can give the user the possibility of subscribing to some of these with his email address. For that, we just need to use the ARN and the email address of choice:

```
import com.amazonaws.services.sns.AmazonSNS;
import com.amazonaws.services.sns.model.SubscribeRequest;
import com.amazonaws.services.sns.model.Topic;

// ...

// get sns service
AmazonSNS snsService = ...

// obtain the email of the user who wants to subscribe
String address = ...

// obtain the topic chosen by the user (from the list obtained above)
Topic topic = ...
String topicARN = topic.getTopicArn();

// subscribe the user to the topic with protocol = "email"
snsService.subscribe(new SubscribeRequest(topicARN, "email", address));
```

The user will then receive an email message requesting confirmation of this subscription.

Sending notifications on a topic

It's also very easy to send notifications about a given topic. Just use the topic ARN and provide a `String` for a message and an optional subject:

```
import com.amazonaws.services.sns.AmazonSNS;
import com.amazonaws.services.sns.model.PublishRequest;
import com.amazonaws.services.sns.model.Topic;

// ...

AmazonSNS snsService = ...

// obtain the topic chosen by the user (from the list obtained above)
Topic topic = ...
String topicARN = topic.getTopicArn();

snsService.publish(new PublishRequest(topicARN,
    "A server is in trouble!", "server alarm"));
```


www.it-ebooks.info

In Short

In this chapter, we used several real-life examples of usage of three services—SQS, SimpleDB, and SNS. In most cases, these services help in decoupling the components of your system. They introduce a new way for the components to communicate, and allow for each component to scale independently.

In other cases, the benefits lie in the new functionality these services enable, like in the example of priority queues with SQS for processing files. Ease of use, high availability, and the absence of setup and maintenance overhead are usually good reasons to use the SimpleDB data store. SNS can be used to communicate events among different components. It provides different types of endpoints, such as email, HTTP, and SQS.

We used three of our sample applications, Kulitzer, Marvia, and Decaf, to show code examples in three different programming languages—Ruby, PHP, and Java.

In the next chapter, we'll be making sure our infrastructure stays alive.

www.it-ebooks.info

CHAPTER 5

Managing the Inevitable Downtime

With smaller infrastructures, downtime is inevitable. Small infrastructures are different from large ones in that you basically do not have the means to get rid of all your single points of failure. With physical systems, downtime due to hardware failure is a big problem, and waiting for replacement parts is a nerve-wracking experience. And if you have the funds to stock replacements, you can just as well put them in production and remove your single points of failure. With a cloud infrastructure, you don't have this problem; you can replace most of your assets whenever you want. This characteristic is central in our approach to managing small infrastructures. You might say we plan to fail.

As in hardware infrastructures, in cloud infrastructures, failing hardware is one cause of trouble. Insufficient capacity is another. In this chapter, we will look at how to measure your system. Is the app up or down? Are the disks over capacity? Is the load breaching expected thresholds? What is the CPU utilization of the RDS instance? We will show you how to monitor your systems from the inside and the outside. We'll take a close look at CloudWatch. We will describe the tools you can use to understand what your system is doing. With this understanding, you can manage your infrastructure if it goes down, or just help it cope with increasing demands. Having limited resources is an opportunity to optimize your system and get the most out of your hardware. This phase of managing your infrastructure is instrumental for the entire lifecycle—every optimization will pay itself back many times when you have to scale out with more instances.

There are two leading principles for this type of infrastructure:

- Minimize downtime by planning for quick recovery
- Optimize the app by restricting infrastructure scaling

We do this by measuring, understanding, and improving!

www.it-ebooks.info

In this chapter, we use yet another of our partners: Directness. Directness helps companies connect with their customers. Directness has a couple of web apps, each of which runs on a separate (medium) instance. The instances are built as described in Chapter 2. The databases are all on one RDS instance. Because Directness is an information-intensive company, we chose to run a multi-AZ small RDS instance; we can't afford to lose anything. And since its business is successful, it is growing fast and continuously developing new apps and adding new features to existing applications. We will show you the tools we can use to measure and understand the Directness infrastructure, and sketch the improvement strategies we propose to implement.

Measure

If you want to help your app stay alive, you need to know what is going on—you need to measure. At minimum, you need to perform measurements to verify whether your system is available. However, with this type of monitoring, when something happens you are already too late—your app is already down. There are many standard ways for a system to fail that you want to be notified about. But alerting on things that fail is only half of our work. We also need to measure to find improvements—we want to *keep* this system alive.

Up/Down Alerts

The most basic, and indispensable, monitoring you can have is up/down monitoring. This used to be quite expensive, so many people built systems themselves based on Cacti or Nagios, integrating the system with an SMS service. Not too long ago, Pingdom became available. This monitoring service is sufficient for determining if your application is running or not. You can monitor ports directly to see if a service like Postfix (SMTP) is still listening, or you can monitor pages. Pingdom is well featured, accepting self-signed certificates, for example. Figure 5-1 shows a screenshot of Pingdom when adding a check for Recommendi, one of the web applications of Directness.

A cheaper alternative is Decaf for EC2, or Decaf Monitor (Figure 5-2). This is an Android application offering almost the same functionality as Pingdom, but for free. The drawback is that mobile networks are not 100% reliable yet. We use both services, mainly because several of our clients have mobile applications, so with Decaf, we can test from the same mobile networks used by the client applications.

Monitoring on the Inside

In addition to up/down monitoring, you probably want to safeguard yourself from other situations as well. You'll want to check on things like load or memory utilization, and you certainly want to determine whether your disks are getting full.

www.it-ebooks.info

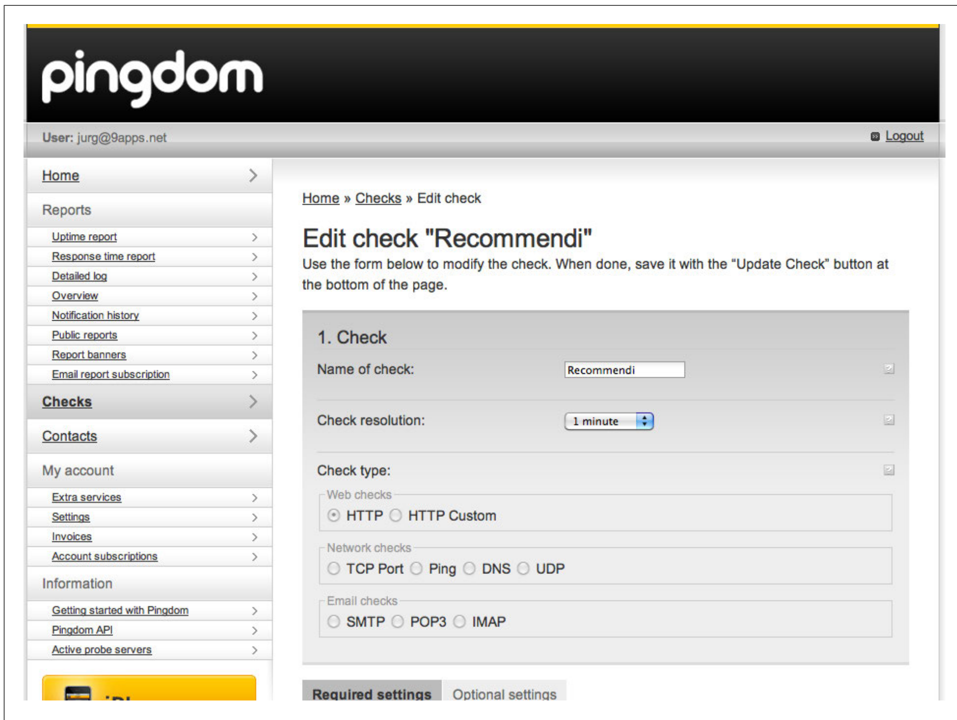


Figure 5-1. Pingdom monitoring Recommendi, one of the Directness web applications

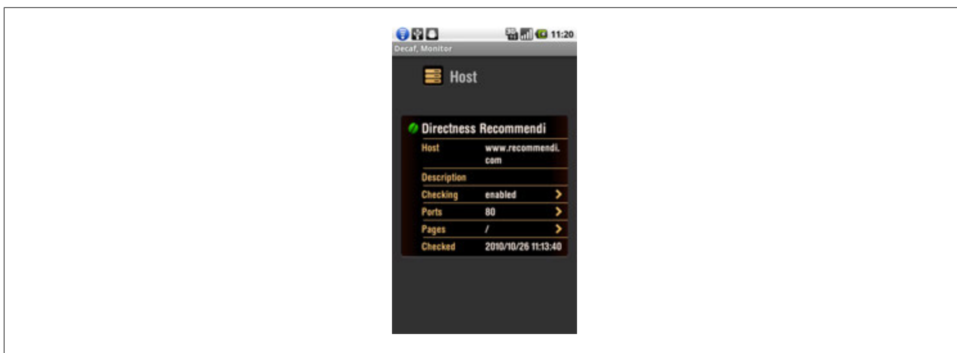


Figure 5-2. Decaf Monitor; monitoring Directness Recommendi

For POSIX systems, there is the excellent Monit. The main advantage of Monit is that it is cloud-ready. Perhaps it is unintentional, but the way Monit is designed to work is perfect for elastic infrastructures. When we use Auto Scaling, we don't know which instances are running. This makes it very difficult to monitor them and to understand what is going on. Remember that we had to implement self-deployment strategies when using Auto Scaling, and we have to do the same with monitoring.

www.it-ebooks.info

Monitoring solutions like Nagios and Cacti use SNMP and are pure client-server. There is a monitoring server asking the client (your server) what is going on. Monit is radically different. Monit is installed on the system, and can notify a server (called MMonit) or send you an email message in case of emergency. The drawback is that you won't know if the system fails completely, but we have other tools for that. See Figure 5-3.

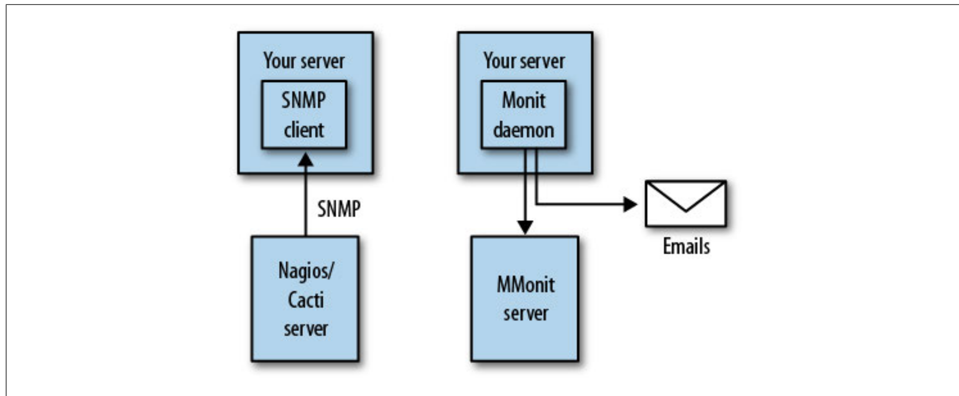


Figure 5-3. Nagios/Cacti versus Monit

In addition to monitoring disk, CPU, and load, we can also monitor many other things with Monit. We can monitor services like Apache in several ways, for example. And we can have Monit do basic repairs with the start/stop service if something fails. We regard restarting a service as the worst way of making your system work, but it is very convenient.

Here is a Monit configuration file that checks the load on the server, directory permissions, and disk space, and verifies that ssh and apache executables have not been modified and are running normally. These checks are done every minute. If load or disk space surpasses some limits, or if we find some changes in key files or directories, we send email messages to ourselves for taking action. Notice that we don't use a MMonit server; we only send messages for alerting:

```
##
##

## check every minute
set daemon 60
set logfile syslog facility log_daemon

## configure smtp server to send alerts by mail
set mailserver smtp.gmail.com port 587 username "yourname@gmail.com"
    password "yourpassword" using tlsv1
set alert hanus@9apps.net
set alert jurg@9apps.net

## Services
```


www.it-ebooks.info

```
  ## check number of processes in the run queue
  check system www.recommendi.com
    if loadavg (1min) > 4 then alert
    if loadavg (5min) > 2 then alert
    if memory usage > 70% then alert

## MOUNTS

  ## check some key directories for permissions
  ## and disk space
  check filesystem root with path /dev/sda1
    if failed permission 660 then alert
    if failed uid root then alert
    if failed gid disk then alert
    if space usage > 80% for 5 times within 15 cycles then alert
    if space usage > 99% then stop

  check filesystem mnt with path /dev/sda2
    if failed permission 660 then alert
    if failed uid root then alert
    if failed gid disk then alert
    if space usage > 80% for 5 times within 15 cycles then alert
    if space usage > 99% then stop

  check filesystem www with path /dev/sdf
    if failed permission 660 then alert
    if failed uid root then alert
    if failed gid disk then alert
    if space usage > 80% for 5 times within 15 cycles then alert
    if space usage > 99% then stop

## PROCESSES

  ## check that ssh executable has not been
  ## altered and permissions are correct
  check file ssh_bin with path /usr/bin/ssh
    alert security@9apps.net on
        {checksum, timestamp, permission, uid, gid}
             with mail-format {subject: SSH Alaaarrm! on www.recommendi.com}
    if failed checksum and
       expect the sum b26174c7f4d2d7eebb68dbed3e2b884e then unmonitor
    if failed permission 755 then unmonitor
    if failed uid root then unmonitor
    if failed gid root then unmonitor
    group ssh

  ## if we can't connect to port 22,
  ## then restart the service
  check process ssh with pidfile /var/run/sshd.pid
    start program = "/etc/init.d/ssh start"
    stop program  = "/etc/init.d/ssh stop"
    if failed port 22 then restart
    if 2 restarts within 3 cycles then timeout
    depends ssh_bin
```


www.it-ebooks.info

```
  group ssh

## check that the apache executable has not been
## altered and permissions are correct
check file apache2_bin with path /usr/sbin/apache2
  alert security@9apps.net on
      {checksum, timestamp, permission, uid, gid}
           with mail-format {subject: Apache2 Alaaarrm! on www.recommendi.com}
  if failed checksum and
     expect the sum fb521a6d6da8350948ab64b01e3464c2 then unmonitor
  if failed permission 755 then unmonitor
  if failed uid root then unmonitor
  if failed gid root then unmonitor
  group www

## if we can't connect to port 80 on localhost
## then try restarting apache
check process apache with pidfile /var/run/apache2.pid
  start "/etc/init.d/apache2 start"
  stop  "/etc/init.d/apache2 stop"
  if failed host localhost port 80
      protocol HTTP request "/" then restart
  if 5 restarts within 5 cycles then timeout
  depends on apache2_bin
  group www
```

Monitoring on the Outside

We have our systems in place for when things break or are about to break. But we want to know more. With physical infrastructures, you could use the same systems as previously mentioned, Nagios or Cacti (and there are probably many more great ones out there). Monit also has a tool for measuring how your infrastructure is doing. Basic usage of these services includes measuring CPU, network, and disk, and the data is shown in nice graphs over time.

AWS has something similar: CloudWatch. With CloudWatch, you can watch EC2 and RDS instances, EBS volumes, ELBs, etc. CloudWatch graphs are present in the AWS Console, and can be very helpful in investigating your system's performance. For all individual assets, you can see the CloudWatch metrics in graphs. These graphs are interactive, providing you with a fine-grained way of drilling down in the performance of your assets (Figure 5-4).

There are still some things missing in the Console, however. One of those things is the ability to aggregate measurements over multiple assets. You can get aggregate measurements with the command-line tools, but as a raw list of numbers—no charts are generated for you. You can get an average CPU utilization of a group of instances in a particular security group, for example. You can also get CPU utilization, disk in/out, and network in/out aggregated over your entire account. This is invaluable, as it shows how your app is doing at a glance. You know when to pay attention if you follow this.

www.it-ebooks.info

With CloudWatch, you can get this information (using the command-line tools), but you can’t view it graphically in the Console.

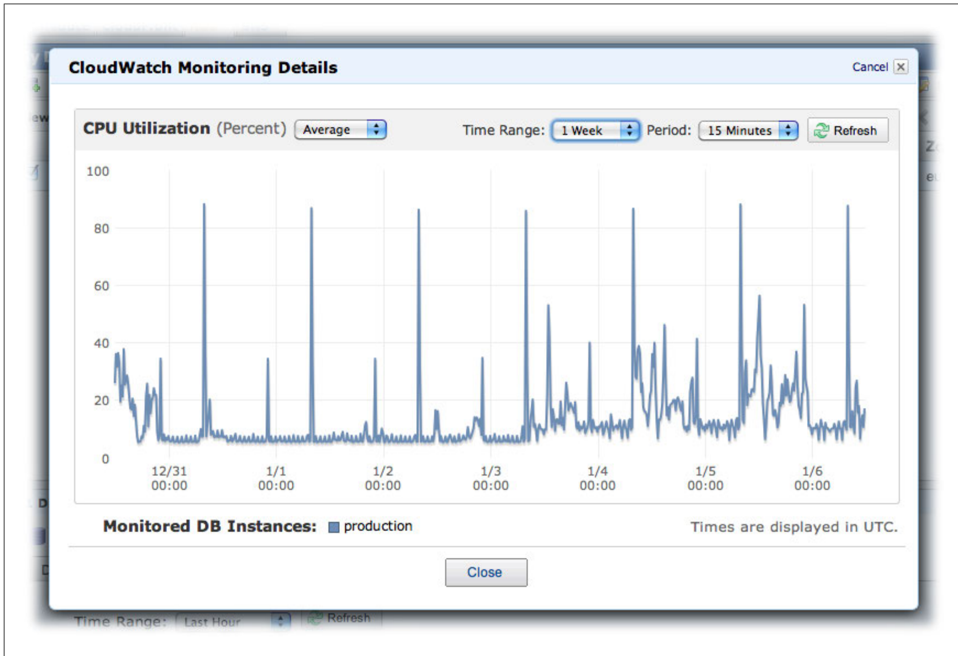


Figure 5-4. CloudWatch chart showing CPU utilization on an RDS instance

We have implemented this type of monitoring in Decaf, which has a widget that shows overall CPU utilization, disk, and network, giving you a way to look into your app’s performance (Figure 5-5).

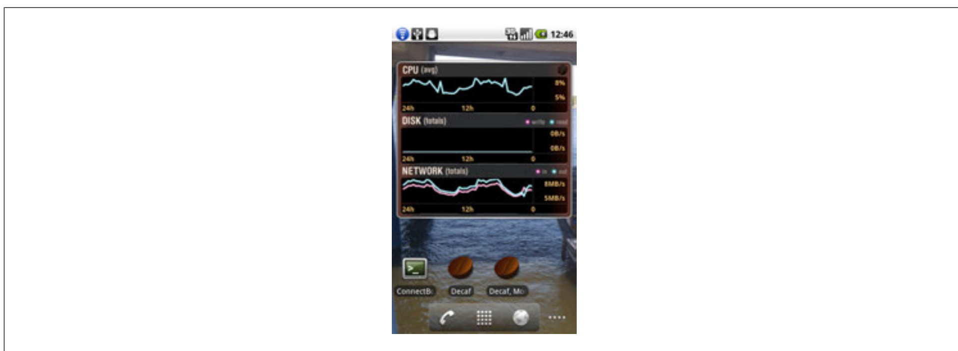


Figure 5-5. Decaf EC2 widget

www.it-ebooks.info

CloudWatch

CloudWatch lets you watch EC2 instances, EBS volumes, ELBs, and RDS instances. For EC2 instances, enabling detailed CloudWatch incurs a small running cost on top of what you normally pay. You can enable/disable this on a per-instance basis, at launch or while running. If you use Auto Scaling, CloudWatch is enabled by default.

CloudWatch is not easy to use, but it is very versatile. It is important to note that CloudWatch data is only kept for two weeks, and is aggregated over periods of 60 seconds. This makes it unsuitable for long-term trending, but it's enough to measure shifts in application behavior and usage. If you need long-term trending, you can gather the data you need regularly and store it in your own database.

There are a couple of concepts you need to be familiar with when working with CloudWatch from the command line or using the API:

Dimension
: The dimension specifies which asset (group) you want to measure. It can be an `InstanceID`, but you can use it to aggregate over other aspects of your infrastructure. You can do many interesting things with dimensions: you can monitor all instances from an autoscaling group in a particular availability zone, or monitor all instances from a particular instance type or image. For example, to get the average CPU utilization of all instances in an autoscaling group (used by Auto Scaling itself) over periods of 15 minutes, you can execute the following:

```
$ mon-get-stats CPUUtilization \
    --dimensions="AutoScalingGroupName=app-server-as-group-1" \
    --statistics Average \
    --namespace="AWS/EC2" \
    --unit Percent \
    --start-time 2010-10-23T22:00:00 \
    --end-time 2010-10-24T22:00:00 \
    --period 900
```

Metric
: A metric is what you want to measure. In the previous example, the measure is `CPUUtilization`. A metric has a particular unit in which it is stored, for example percentage or bytes. A metric can have certain dimensions (e.g., for `CPUUtilization` for EC2 instances, you can choose an `InstanceID` as dimension or an autoscaling group, as in the previous example), and it is part of a *namespace*.

Namespace
: A namespace is the source of the measurements. There are four namespaces at this moment: EC2, EBS, ELB, and RDS. In the example, we query information from the AWS/EC2 namespace.

www.it-ebooks.info

Statistic
: There are four statistics available: minimum, maximum, average, and sum. A statistic has a one-minute value that can be aggregated over a period consisting of a number of minutes. We asked for averages over 15-minute slots between October 23, 22:00 and October 24, 22:00.

Unit
: Every measure has a particular unit attached. Some measures have only one unit, some have different units. If you don't specify the unit, you get everything. For `CPUUtilization`, there is only one, so we could have omitted the `Percent`. But network or disk can be measured in bits or bits/second (or bytes and bytes/second).

ELBs. ELBs have a couple of interesting CloudWatch metrics. You can see the `RequestCount`, for example, to determine how much the load balancer is getting hit, and you can also see the latency. In the following example, we'll see the number of requests to the load balancer over periods of 15 minutes during the given day. Notice that we use `Sum` for statistics. The second example shows the average latency of the load balancer over periods of 15 minutes during the given day:

```
$ mon-get-stats RequestCount \
    --dimensions="LoadBalancerName=production" \
    --statistics Sum \
    --namespace="AWS/ELB" \
    --start-time 2010-10-23T22:00:00 \
    --end-time 2010-10-24T22:00:00 \
    --period 900

$ mon-get-stats Latency \
    --dimensions="LoadBalancerName=production" \
    --statistics Average \
    --namespace="AWS/ELB" \
    --start-time 2010-10-23T22:00:00 \
    --end-time 2010-10-24T22:00:00 \
    --period 900
```

Region. The measurement we use to visualize an entire region in Decaf is an informative way of watching the strain on your app in that particular region. There is no way to aggregate CloudWatch data over regions. If you want to see your average CPU utilization in the region `eu-west-1`, you can use the following:

```
$ mon-get-stats CPUUtilization \
    --statistics Average \
    --namespace="AWS/EC2" \
    --start-time 2010-10-23T22:00:00 \
    --end-time 2010-10-24T22:00:00 \
    --period 900 \
    --region eu-west-1
```


www.it-ebooks.info

Understand

Every application is different—they all have their own particular signatures. When monitoring your application regularly with the tools we have described, you will get to know that signature. You will know when to expect your CPU to spike because of a reporting cron job, or what the normal traffic graph is. And anything out of the ordinary is worth a closer look.

There is a wealth of information available through CloudWatch, and with some effort, you can correlate different graphs. What you can't read from the graphs is *why* traffic spiked, and it's not always the application that is misbehaving. It's important to talk to everyone involved so you can figure out what happened, but more importantly, you will start to learn what to expect. After all, before you can conclude something is fishy, you need to have expectations.

Why Did I Lose My Instance?

The most dramatic failure is downtime. If your application runs on one instance, that instance is the most vulnerable part of your infrastructure. Luckily, we built our instance in such a way that we can easily launch another one. But after the fact, it is important to know what happened.

If your instance is still around, you can investigate what went wrong. If your instance is not there anymore, or if the Console reports it running but you can't access it, it is more difficult. In these cases, you can politely ask in the AWS forums if there was a hardware or network failure. Most of the time you get an answer, eventually (Figure 5-6). Hardware and network failures happen, but it is good to know whether to suspect your own app, or just be happy you built a good infrastructure that allows you to launch a new instance within minutes.

Spikes Are Interesting

Looking at the RDS instance CloudWatch graphs from Directness, we immediately see spikes (Figure 5-4). Some of the spikes are regular, but there are a number that deviate from the expected behavior. The graph of one hour with one-minute intervals during the day reveals that these spikes are more the norm than the exception (Figure 5-7). This shouldn't be the case, so we try to find the reasons why this might be happening.

The applications using this RDS instance used to run on MySQL servers either on dedicated hardware or sharing the same instance. As described in Chapter 2, RDS has some peculiarities that require you to be careful with disk-intensive operations. For Directness, for example, we are using views for user reports.

www.it-ebooks.info

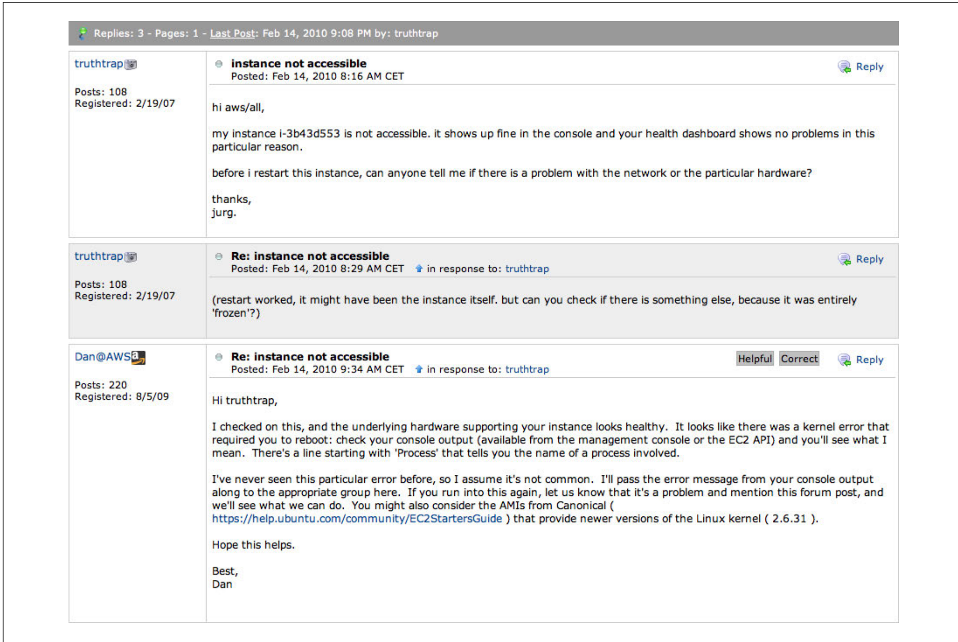


Figure 5-6. Asking for help in the AWS forum

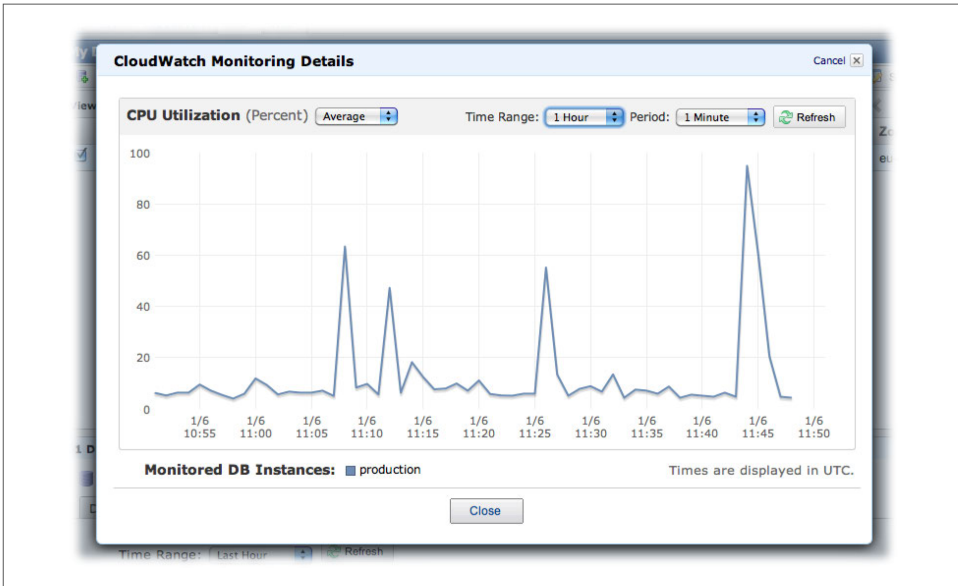


Figure 5-7. CPU utilization on RDS instance during one hour

www.it-ebooks.info

These views were not too much of a problem at the beginning, because we had capacity to spare on our RDS instance. We had to make sure the periods of our reports were not too big so that the overall performance of the applications would not be degraded.

But we can now see that some spikes eat up all our RDS capacity. This means we have to reevaluate the use of the database. You can see how looking at spikes in our CloudWatch charts can help us pinpoint problematic areas.

Predicting Bottlenecks

Apart from spikes, we always try to get a feel for the next bottleneck. What exactly is normal behavior is totally dependent on the application, but we always have presumptions. One of the presumptions, for example, is that we should be able to run a number of application servers per RDS instance. So we regularly check the utilization of our EC2 instances and compare them to the utilization of the RDS instance.

For Directness, we see that the utilization of the EC2 instances corresponds to the utilization of the RDS instance; the usage pattern is about the same. We can conclude that, for the most part, there are no obvious mistakes in the application. If there are misbehaving queries—for example, queries that don't make use of indexes or make heavy use of the filesystem—we would see RDS utilization skyrocket while EC2 instance utilization remains flat.

What we do see is that the RDS instance is running between 20% and 40% CPU utilization. At the same time, the average CPU utilization of the EC2 instances hardly exceeds 10%. This means we can't even run a fully utilized application server on one small RDS instance. If the EC2 instances were to have more load, say, 30% CPU utilization, the RDS instance wouldn't be able to handle it. This is not what we want, and is reason enough to try to understand what is going on.

Improvement Strategies

Considering we don't have the luxury of unlimited funds, we've already accepted the possibility of occasionally degraded performance or even unavailability. If performance degrades too much, we always have a way out by scaling up, increasing the size of the box. This is easy, so we have the opportunity to push our luck a bit. We can try to find the real limits of our small infrastructure by being deprived of resources, and we'll only scale up when we really have no other option. By then, we will probably be ready to start planning for some elasticity in our infrastructure.

Benchmarking and Tuning

There are two types of performance measurements: absolute and relative. We think that measuring absolute performance is only necessary for something like management

www.it-ebooks.info

software for a nuclear power plant. So, most of the time, we are more interested in relative performance.

In Internet apps (web, mobile, etc.), there are always two sides to performance: the client side and the server side. You can help the client by using CloudFront, for example. For the rest, it basically comes down to using the checklists provided by excellent tools like FireBug or the developer tools in Chrome. On the server side, however, we can help a lot. If your server is not too busy, performance is usually OK (if you followed the checklists, of course). But if your server is getting busier, performance can degrade quickly.

It is relatively easy to test this kind of performance. With tools like Apache Bench or JMeter, you can hit your infrastructure pretty hard and see what happens. In-depth tuning is definitely not within the scope of this book—many good books have been written on this topic. But we follow these general guidelines in optimizing/tuning your instances:

- Make sure your services (Apache, PostgreSQL, etc.) stay within the limits of the instance (e.g., memory should be limited to avoid processes that cause swapping).
- Utilize as much of your assets (memory, CPU) as possible.

Apart from these guidelines, we always look for best practices. For PHP, for example, we can recommend Alternative PHP Cache, or APC. This increases request throughput a lot. But at this stage of your application, you shouldn't be doing too many exotic performance improvements. Try to get to good utilization of your system's resources, and spend the rest of your time developing a clean app.

The Merits of Virtual Hardware

With size comes abundance of spare resources. If you plan for 20% over capacity (20% of the server capacity standing ready), that doesn't account for much when you have five servers. But when you have 100,000 servers, it is a lot. The actual capacity of AWS is a well-kept secret, but we rarely get the message "insufficient capacity." And when we do, launching the same instance in another zone usually does the trick.

Replace hardware

In 2008, Google participated in a report on computer memory. It drew a number of interesting conclusions, including "a surprisingly strong and early effect of age on error rates." All physical systems are under some form of strain—logically, error rates will increase over time. The expensive way to reduce the risk associated with this phenomenon is to buy more durable components. And this is normally the way to design critical systems: to buy the most durable components you can afford.

Google was one of the first to take a different approach. It has many machines designed with failure in mind, opening up the possibility of using cheap off-the-shelf compo-

www.it-ebooks.info

nents. For a long time, this was also a domain for the wealthy, but not anymore. With AWS, you can build your infrastructure around the principle of using cheap hardware and replacing it when it fails: you can replace your hardware any time you want, at no cost whatsoever. You probably won't always get new hardware, but you definitely increase your chances of getting new or replenished hardware. You certainly will not be assigned an instance on hardware designated for replacement.

How can you apply this knowledge? If you do maintenance, you probably have to build a new image. You can create images without stopping your instance, but it does take them down for a brief period of time. If you stop your instance explicitly, create an image, and launch a new instance, your chances of getting new hardware are quite high. Just a reboot probably will not replace your physical hardware.

Scaling up

One of the most valuable merits of AWS is that it allows you to scale up and down painlessly. It doesn't take much to replace your RDS or EC2 instance with a bigger version. RDS, especially, shows the ease of this feature. It doesn't require any tuning or benchmarking, as AWS completely implements optimizing the database for the underlying hardware.

> Although tuning an RDS instance is not recommended, you can play around with certain parameters. You can, for example, enable query caching and set the size of the cache to your liking. But if you do so, you have to adjust these parameters when you scale up or down.

For Directness, this means we can lessen the burden of an overused RDS instance by scaling up to a bigger one. Of course, this is more expensive, but it buys us time to do necessary optimizations in the application layer. We can implement caching, for example, or refactor our reporting. If we see the RDS utilization has dropped to normal levels, we can scale down again.

Auto Scaling group of one

We described Auto Scaling from the perspective of elastically handling traffic by only using the instances we need. But Auto Scaling also replaces dysfunctional instances, if necessary, so you can use Auto Scaling to guard a group of one instance. We don't really use this particular feature, because fixing our applications by deploying a new instance is easy. But if you want, you can improve your performance with Auto Scaling.

In Short

In this chapter, we looked at different ways of watching your infrastructure to make sure it's up at all times. We talked about different tools that can check if your application

www.it-ebooks.info

is available and monitor certain variables such as memory or CPU usage, latency, and number of requests. These tools—Pingdom, Decaf, Monit, CloudWatch—provide ways of configuring alarms when certain conditions are met. That your system is not online anymore is an obvious case, but you will also want to check, for example, if your RDS instance is using its CPU at more than 60% or if the latency is higher than 0.5 seconds.

Looking at CloudWatch charts or data output can help you find undesirable situations. You can find unusual peaks in usage that can point you to necessary improvements. Or you can see which part of your infrastructure is or will become a bottleneck.

Tuning your infrastructure can be tricky, since you always need to make sure your software uses physical assets like memory and CPU within some limits. But you also want to optimize and use those assets as much as possible before scaling. We'll be diving deeper into this topic in the next chapter.

www.it-ebooks.info

CHAPTER 6
Improving Your Uptime

We now have an application that uses the techniques outlined in Chapter 3 for scaling gracefully. This application has become very resilient—the single points of failure have been eliminated—and as a consequence, we sleep much better at night. The game has changed: we have our downtime under control. It's time to start working on improving the quality of our uptime.

The essence of improving our infrastructure comes down to the following:

- Handling the acceleration of traffic (increase/decrease)
- Optimizing utilization

Now that we have implemented our infrastructure to handle scale, the peaks are no longer our immediate problem. Instead we have to worry about the acceleration in traffic, *how* we climb up and down those slopes.

Our infrastructure is probably also growing. We are already more efficient because we scale automatically. If we can increase the traffic individual instances can handle, we will optimize utilization of our assets. And that is exactly the underlying principle of cloud computing: minimize waste by optimizing utilization.

In the upcoming sections, we will look at some ways to measure and monitor the use of an AWS infrastructure. We will then analyze this information and show how you can use it to tune your components to optimize use of resources and become more resilient to changes in traffic.

Measure

As we have shown in the previous chapter, Amazon CloudWatch is a very versatile monitoring tool. You can measure all sorts of things. But more importantly, you can aggregate by using dimensions. Dimensions give us a way to measure groups of instances, an availability zone in combination with an ELB, or an entire region. We can use aggregation over a region to get an overall feel for what is going on.

www.it-ebooks.info

There are several dimensions available in the different namespaces. Currently, there are four different namespaces: EC2, ELB, RDS, and EBS. As far as we can see, the only dimensions available for RDS and EBS are single assets, either an RDS database identifier or a volume identifier.

The metrics and dimensions are important. We have taken the lists from the Amazon CloudWatch developer guide. They do change, so make sure to check.

EC2

At the moment, you can measure the CPU, disk, and network activity, and you can either watch one particular instance or aggregate the measurements by AMI, autoscaling group, and instance type.

Dimensions

`ImageId`
: This is the most useful dimension for the purposes of measuring our application at this stage. It allows us to measure the aggregated performance of all web servers, assuming they all originate from the same AMI, of course.

`AutoScalingGroupName`
: This dimension is used by autoscaling itself to trigger scale activities. We want to use it the same way as we monitor load, and be alerted about certain thresholds.

`InstanceId`
: This is the dimension used for instances in the AWS Console. It provides measures about a single instance.

`InstanceType`
: Measuring an instance type is interesting in the context of instance limitations. An infrastructure component can be bound by network or disk activity rather than the CPU. You can use this to measure how certain kinds of instances are doing in that regard. Unfortunately, this dimension cannot be combined with the ImageId dimension. The only two dimensions that can be used together are the ELB dimensions (see below).

Metrics

`CPUUtilization`
: The percentage of allocated EC2 computation units that are currently in use on the instance. This metric identifies the processing power required to run an application upon a selected instance.

`DiskReadOps`
: The number of completed read operations from all disks available to the instances. This metric identifies the rate at which an application reads a disk. You can use this to determine the speed at which an application reads data from a hard disk.

www.it-ebooks.info

`DiskWriteOps`
: The number of completed write operations to all hard disks available to the instance. This metric identifies the rate at which an application writes to a hard disk. You can use this to determine the speed at which an application saves data to a hard disk.

`DiskReadBytes`
: The number of bytes read from all disks available to the instance. You can use this metric to determine the volume of the data the application reads from the hard disk of the instance. This can help you determine the speed of the application.

`DiskWriteBytes`
: The number of bytes written to all disks available to the instance. Use this metric to determine the volume of the data the application writes onto the hard disk of the instance. This can help you determine the speed of the application.

`NetworkIn`
: The number of bytes received on all network interfaces by the instance. This metric identifies the volume of incoming network traffic to an application on a single instance.

`NetworkOut`
: The number of bytes sent out on all network interfaces by the instance. This metric identifies the volume of outgoing network traffic to an application on a single instance.

ELB

In the ELB namespace you can measure the latency, the number of requests, and the number of healthy and unhealthy instances. You can get this information either for the traffic in one load balancer or for the traffic in an availability zone.

Dimensions

`LoadBalancerName`
: This dimension is only available in the command-line tools—not yet in the AWS Console. This is an important dimension. Metrics associated with it are `Latency` and `RequestCount`, among others. It provides another way to measure a group of instances, this time the group behind a load balancer.

`AvailabilityZone`
: This dimension compares the different availability zones in use behind an ELB. Remember that load balancers divide the traffic equally over different availability zones. If you have a mixed set of instances (you can have one of the instances performing other tasks), you can look for differences.

www.it-ebooks.info

Metrics

`Latency`
: The time taken between a request and the corresponding response as seen by the load balancer.

`RequestCount`
: The number of requests processed by the load balancer.

`HealthyHostCount`
: The number of healthy instances. Both ELB dimensions, `LoadBalancerName` and `AvailabilityZone`, should be specified when retrieving `HealthyHostCount`.

`UnHealthyHostCount`
: The number of unhealthy instances. Both ELB dimensions, `LoadBalancerName` and `AvailabilityZone`, should be specified when retrieving `UnHealthyHostCount`.

RDS

In the RDS namespace, in addition to metrics for measuring CPU and disk usage, we find latency, throughput, number of database connections, storage usage, log size, and read replica synchronization lag. You can choose to measure one specific RDS instance, all the instances of a particular class, or the instances that use a certain engine.

Dimensions

`DBInstanceIdentifier`
: This dimension filters the data you request for a specific database instance.

`DatabaseClass`
: This dimension filters the data you request for all instances in a database class. For example, you can aggregate metrics for all instances that belong to the database class `db.m1.small`.

`EngineName`
: This dimension filters the data you request for the identified engine name only. For example, you can aggregate metrics for all instances that have the engine name `mysql`.

Metrics

`ReplicaLag`
: The amount of time a read replica DB instance lags behind the source DB instance.

`BinLogDiskUsage`
: The amount of disk space occupied by binary logs on the master.

`CPUUtilization`
: The percentage of CPU utilization.

www.it-ebooks.info

`FreeStorageSpace`
: The amount of available storage space.

`DatabaseConnections`
: The number of database connections in use.

`ReadIOPS`
: The average number of disk I/O operations per second.

`WriteIOPS`
: The average number of disk I/O operations per second.

`ReadLatency`
: The average amount of time taken per disk read operation.

`WriteLatency`
: The average amount of time taken per disk write operation.

`ReadThroughput`
: The average number of bytes read from disk per second.

`WriteThroughput`
: The average number of bytes written to disk per second.

Using Dimensions from the Command Line

Measuring a group of instances launched from a particular image is quite straightforward. It would be nice if the AWS Console would implement this particular feature, although you can select multiple instances to be graphed. So to get the average CPU utilization of all the instances launched from `ami-30360344` in periods of 15 minutes, you can do the following:

```
$ mon-get-stats CPUUtilization \
    --dimensions="ImageId=ami-30360344" \
    --statistics Average \
    --namespace="AWS/EC2" \
    --unit Percent \
    --start-time 2010-11-04T11:00:00 \
    --end-time 2010-11-04T13:00:00 \
    --period 900
```

To get the average CPU utilization of all large instances, you do this:

```
$ mon-get-stats CPUUtilization \
    --dimensions="InstanceType=m1.large" \
    --statistics Average \
    --namespace="AWS/EC2" \
    --unit Percent \
    --start-time 2010-11-04T11:00:00 \
    --end-time 2010-11-04T13:00:00 \
    --period 900
```

The output is not very graphical, to say the least, but it does give you a flexible way to read out the performance of your application components.

www.it-ebooks.info

Alerts

Immediate reporting is not quite as necessary as it when the system is down, but we do like to receive notices of particular events, for example, that "average CPU utilization is above 40% for the last 5 minutes." A message describing the event is enough; we can then investigate it later in time (remember, we have two weeks of CloudWatch data to work with).

We already had Monit, which is capable of performing checks on web pages (repeatedly) and reporting on failures. And we had a PHP script to execute these checks, passing the dimensions, metrics, and upper threshold through URL parameters.

But, just one month before our deadline for this book, AWS added alerts to CloudWatch. This was expected, and it makes alerting much more integrated and powerful.

The kind of events that are of interest to us look like this:

- `CPUUtilization` of the RDS instance exceeding 60%
- Aggregated average **`CPUUtilization`** of all web instances higher than 40%
- `RequestCount` of ELB more than 75 requests per second

With the CloudWatch API tools, we can add these alerts. Alarms are sent using SNS, so before we start adding the alarms, we have to create an SNS topic. We created a topic called **`cloudwatch-alarms`**, to which we can add email subscriptions (Figure 6-1). Adding alarms for these events is not straightforward, but as you can see, it is very powerful. These are our alarms:

```
$ mon-put-metric-alarm RDS-CPU-60 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 3 \
    --metric-name CPUUtilization \
    --namespace "AWS/RDS" \
    --period 900 \
    --statistic Average \
    --threshold 60 \
    --actions-enabled true \
    --alarm-actions arn:aws:sns:us-east-1:457964863276:cloudwatch-alarms \
    --alarm-description "RDS over 60% for 3 consecutive periods of 15 mins" \
    --dimensions "DBInstanceIdentifier=production" \
    --ok-actions arn:aws:sns:us-east-1:457964863276:cloudwatch-alarms \
    --unit Percent

$ mon-put-metric-alarm WEB-CPU-40 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 3 \
    --metric-name CPUUtilization \
    --namespace "AWS/EC2" \
    --period 900 \
    --statistic Average \
    --threshold 40 \
    --actions-enabled true \
    --alarm-actions arn:aws:sns:us-east-1:457964863276:cloudwatch-alarms \
```


www.it-ebooks.info

```
    --alarm-description "Web servers over 40% for 3 consecutive periods of 15 mins" \
    --dimensions "AutoScalingGroupName=app-server-as-group-1" \
    --ok-actions arn:aws:sns:us-east-1:457964863276:cloudwatch-alarms \
    --unit Percent

$ mon-put-metric-alarm ELB-REQUESTS-75 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 3 \
    --metric-name RequestCount \
    --namespace "AWS/ELB" \
    --period 900 \
    --statistic Average \
    --threshold 67500 \
    --actions-enabled true \
    --alarm-actions arn:aws:sns:us-east-1:457964863276:cloudwatch-alarms \
    --alarm-description \
    "Requests over 75 per second for 3 consecutive periods of 15 mins" \
    --dimensions "LoadBalancerName=production" \
    --ok-actions arn:aws:sns:us-east-1:457964863276:cloudwatch-alarms \
    --unit Count
```

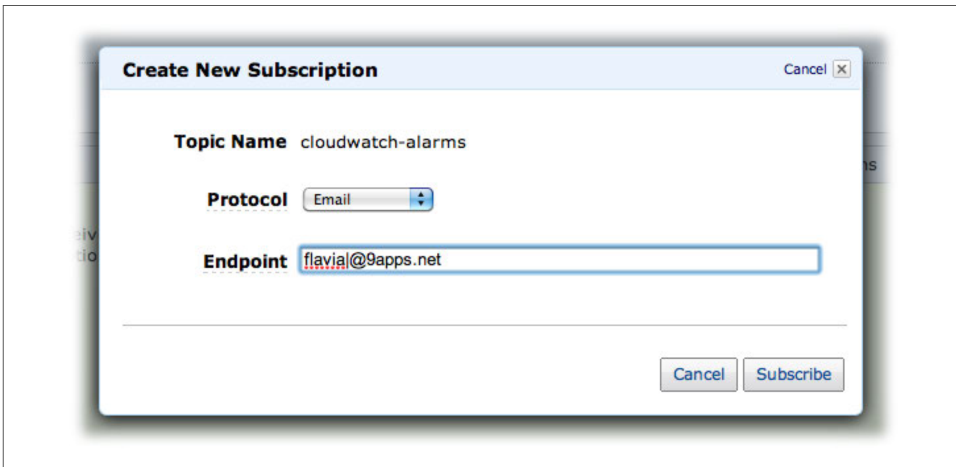


Figure 6-1. Create an SNS topic for sending alarms

When these events occur, nothing is necessarily wrong; we have just passed a certain value for these measurements. In the first case, for example, we know that the RDS instance is being hit too hard relative to the two web instances. We need to optimize the database by looking at the structure, optimizing querying, and implementing caching. But we have chosen to focus our attention on some other pressing matters.

We set the bar of load balancing for this part of our app at around 100 requests per second. We are currently doing 35 requests per second at peak traffic. We will want to take stock of the situation when we are structurally doing 75 requests per second. How is everything holding up? Where are the weak spots?

www.it-ebooks.info

We are not interested in spikes, and we are not interested in occasional high traffic. We query for periods of 900 seconds (15 minutes), three times in a row. We are looking at sustained values for these metrics, not the incidental peaks (Figure 6-2).

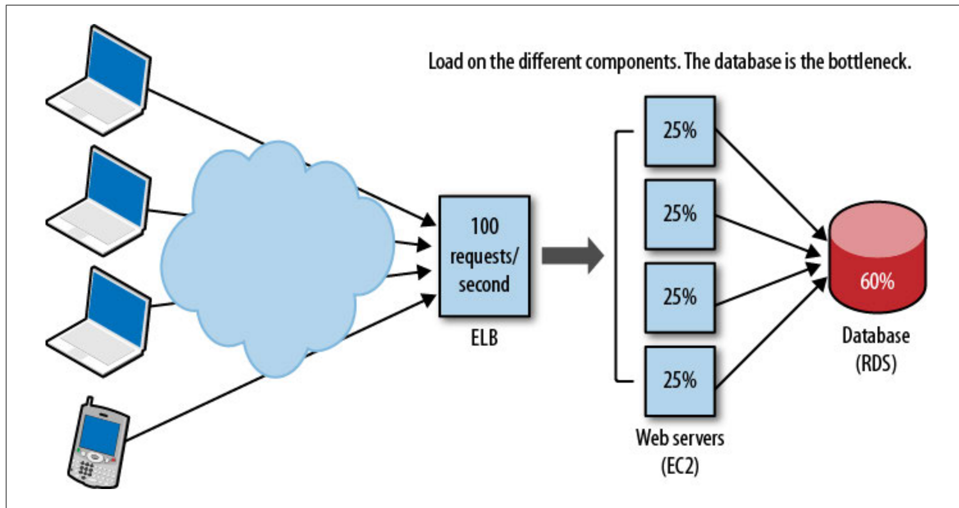


Figure 6-2. Load on the different components; the database is the bottleneck

Understand

Managing small infrastructures is about handling surprises. Measuring (and monitoring) is mostly about being the first to know about a surprise and then using AWS to your advantage and fixing it quickly.

When your infrastructure grows and your single points of failure have been eliminated, there are not many sudden surprises anymore. Operating a larger infrastructure is a more gradual process; it is all about testing expectations.

Setting Expectations

In a large infrastructure, the database is often a weak spot. You are probably well equipped for heavy traffic, but if it gets out of hand, you have nowhere to go. It is not easy to scale, even with services like RDS and the recently introduced read replicas. Memcached, as well, requires changing clients' lists of memcached instances. So what do you expect to get out of your current database ecosystem?

- How many web servers can your DB setup handle?
- How many connections can your DB handle?
- What CPU utilization do you expect with X web servers?

www.it-ebooks.info

There are ways to test all these expectations (or assumptions). Perhaps your app doesn't break when your CPU utilization exceeds 70% with only 4 web servers, but you will probably never reach your expected 10 web servers. You can measure these little things easily, and you always have a period of weeks to look back and understand. This kind of monitoring is critical, but is no longer time-critical. You can do it manually, like a security guard walking his rounds on the premises, or you can help yourself with available tools and techniques. Now you can walk your rounds with a different perspective—understanding what is going on.

Viewing Components

A single web server is not important anymore; when using Auto Scaling, it is not even stable, because it can be terminated if the load is too low. Sure, you want to know if your single instance misbehaves so you can replace it. And you want to optimize each single instance to the max. But the performance of the entire component (group of instances) is more important now.

With the Console, you can get a good view of the performance of a group. Although you can't see only the group, you can select multiple instances (or volumes) at the same time. You get the same interface as for single instances, as shown in Figure 6-3.

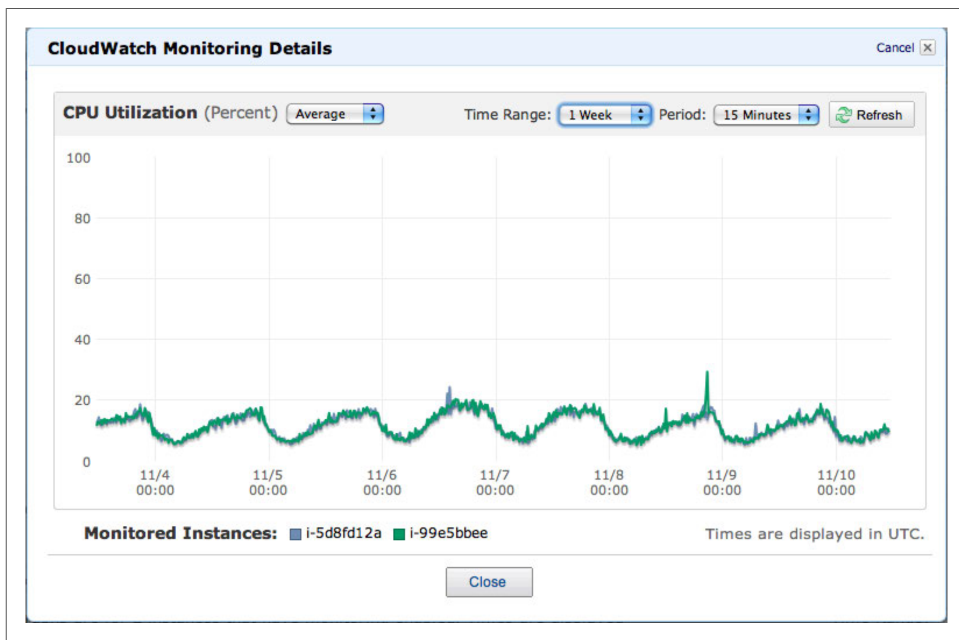


Figure 6-3. CloudWatch; two web servers

www.it-ebooks.info

Improvement Strategies

Our application is running smoothly, and we are optimizing the utilization of our infrastructure. One aspect we can improve upon is redundancy and resilience. In implementing redundancy, we can use the availability zones to increase resilience. Availability zones are physically separate, so launching two instances in separate zones ensures different hardware.

Planning Nonautoscaling Components

Nonautoscaling components will always be there. Databases are the obvious example of nonautoscaling components; even the NoSQL kind of databases do not autoscale.

If scaling is really difficult or takes a lot of time, you want to plan for peak capacity and make sure the system fails gracefully (i.e., doesn't just die and disappoint your users). If scaling manually is easy, you can work closely with your team to figure out where the peaks are, and semiautoscale. You don't have to implement full autoscaling to use AWS elastically.

With the introduction of read replicas in RDS, we can apply this pattern to our MySQL database cluster as well. There is no Auto Scaling for read replicas yet (perhaps by the time you read this, there will be). But if it is not too difficult to add read replicas on the fly, you can manually scale when you expect a lot of traffic.

Tuning Auto Scaling

If you use Auto Scaling, you can use your triggers to your advantage. If you expect dramatic accelerations, you need to have sensitive triggers that kick in early enough to give the new instance time to launch. But the triggers shouldn't be so sensitive that they start scaling up before it's necessary. You can start scaling when the CPU utilization averages 30% for five minutes, which is very sensitive.

If you have higher overall traffic, you will want to start scaling as late as possible. Peaks will be spread out over a number of instances already, so they are not likely to break much. And you want to get the most out of your instances.

If your app is not CPU-bound but network- or disk-bound, you can set your triggers on those metrics too. And you can do exactly the same as with CPU utilization—you can be sensitive or not.

In Short

In this chapter, we looked at ways of optimizing the usage of an AWS infrastructure, and also observing its behavior to be ready to grow it without surprises.

www.it-ebooks.info

We saw that CloudWatch provides a variety of ways to measure the load of EC2 and RDS instances, as well as ELBs. You can choose metrics such as CPU utilization, network I/O activity, and latency, and group the results in different ways, such as per group of instances, type of image, or over a whole region.

You can use CloudWatch together with SNS to send alerts if certain thresholds are surpassed. For example, you can receive an email message when your RDS instance is used over 60% for more than 15 minutes.

With this kind of analysis and measurement, you can be aware of the weaker parts of your infrastructure, and then search for architectural or scaling solutions. For example, do you need to start using read replicas for your RDS instance? You can also see which parts of your infrastructure are being underutilized, and act accordingly. Maybe you need smaller EC2 instances if you see that they are always under 10% CPU utilization.

Finally, you can use this information to tune Auto Scaling thresholds. You get to know your infrastructure more, and you can act accordingly.

www.it-ebooks.info

CHAPTER 7

Managing Your Decoupled System

Something of a congratulation is in order. We've been hard at work, but step by step, we are building a very robust Internet-scale application. We have used a lot of services offered by AWS to make our application better.

The result is that we have several big application components, all managed by their own teams. These application components are in themselves applications, using all the features introduced in Chapters 2 and 3—EC2 instances, RDS databases, ELBs, and Auto Scaling. But they are all glued together by S3, SQS, SimpleDB, and SNS.

These services are very robust and highly scalable, but we still want to know what is going on with our system. Especially in a distributed app like this, with disparate teams, it is very important to know what is happening.

There is no CloudWatch for these services, so we have to figure out how to get to these numbers ourselves. For this, we can turn these services onto themselves.

Measure

We have been calling the AWS services infinitely scalable, but with the disclaimer "for all practical purposes." The question is, when do we reach the end of our practical purposes? That is one of the things we want to measure: the limits of buckets, queues, domains, and groups. These limits say something about the containers themselves.

The other things we want to measure are a bit more vague, and have to do with performance. Queues, domains, and groups do have unique performance characteristics. We want to know how information flows through a queue or an SNS group, for example. Or we would like to know if the attributes of the items in a SimpleDB domain are uniformly distributed or if they are evolving (too much) over time.

www.it-ebooks.info

S3

Of course, we are interested in the S3 performance within our application. Unfortunately, it is hard to get the size of a bucket with a simple operation. We have to iterate over all objects and sum the sizes. Some tools provide this operation, but it takes time for buckets with many objects.

Luckily, S3's only practical limit is on object size (5 TB). We don't have to worry about running out of bucket space. Every month, you get a bill, which is the best way to optimize resource utilization, especially when it comes directly out of your own pocket. If you want to know intermediate numbers, you can go to the account usage page on the AWS portal.

Some people scrape the usage reports and query them to be alerted on usage. We don't need this functionality—monitoring with a frequency of once a month (the bill) is more than enough. If you do need something with a shorter interval, you can also look at `s3cmd`, an excellent tool for working with S3 from the command line.

SQS

SQS has a different role than S3 in our infrastructure. It is a temporary first in, first out (FIFO) buffer with messages that don't need to be processed immediately, but certainly within a reasonable amount of time. What is reasonable is entirely dependent on your application.

We basically want to know the queue size at this moment, alerting us to malfunctions in senders and/or receivers. We are also interested in what we call *queue latency*—not the latency of API requests, but the time between the message being sent and it being picked up for processing. And finally, we want to know *throughput*, the number of messages going through the queue over a particular period.

We would like it if there were a CloudWatch namespace SQS to give us these metrics, but there is not, so we have to do it ourselves. The idea is to build a simple version of CloudWatch for SQS with the help of SimpleDB. We will have the receiving end log their SQS progress to SimpleDB.

We created a Java class that we can call from the receiver to log each message after it's processed. Information about these messages, like sent and processed timestamps, are stored in a SimpleDB domain:

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.TimeZone;
```


www.it-ebooks.info

```
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.AmazonSimpleDBClient;
import com.amazonaws.services.simpledb.model.Attribute;
import com.amazonaws.services.simpledb.model.Item;
import com.amazonaws.services.simpledb.model.PutAttributesRequest;
import com.amazonaws.services.simpledb.model.ReplaceableAttribute;
import com.amazonaws.services.simpledb.model.SelectRequest;
import com.amazonaws.services.simpledb.model.SelectResult;
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClient;
import com.amazonaws.services.sqs.model.GetQueueAttributesRequest;
import com.amazonaws.services.sqs.model.Message;
import com.amazonaws.services.sqs.model.ReceiveMessageRequest;
import com.amazonaws.services.sqs.model.ReceiveMessageResult;


public class SQSLogger {

  private AmazonSimpleDB simpleDB;
  private SimpleDateFormat format;

  public SQSLogger(AWSCredentials credentials) {
    // get the SimpleDB service
    simpleDB = new AmazonSimpleDBClient(credentials);
    format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    format.setTimeZone(TimeZone.getTimeZone("GMT"));
  }

  /**
   * Log an SQS message into the SimpleDB domain "sqs_log".
   * pickedUpTime will be used to calculate latency
   * If the message has been successfully processed
   * we save also the time at which it was processed
   * (deleted from the queue).
   */
  public void logMessage(Message message, Date pickedUpTime, boolean succeeded) {

    String timestamp = message.getAttributes().get("SentTimestamp");

    String sentTimestamp = format.format(new Date(Long.parseLong(timestamp)));

    List<ReplaceableAttribute> attributes = new ArrayList<ReplaceableAttribute>();

    attributes.add(
      new ReplaceableAttribute("SentTimestamp", sentTimestamp, true));

    // All attributes are set to replace=true
    // except for PickedUpTimestamp
    // since the message could be picked up several times
    // until it is successfully processed
    // For latency we need the earliest PickedUpTimestamp
    attributes.add(
      new ReplaceableAttribute("PickedUpTimestamp",
```


www.it-ebooks.info

```
        format.format(pickedUpTime), false));

    attributes.add(
          new ReplaceableAttribute("MessageBody", message.getBody(), true));

    if (succeeded) {
      attributes.add(
        new ReplaceableAttribute("ProcessedTimestamp",
          format.format(new Date()), true));
    }

    // create an item in SimpleDB for this message
    PutAttributesRequest request = new PutAttributesRequest(
      "sqs_log", // simpledb domain name
       message.getMessageId(), // item name
       attributes);

    simpleDB.putAttributes(request);

  }
...

}
```

And this is how we can receive a message and log it using our `SQSLogger`:

```
String accessKey = "AKIAIGKECZXA7AEIJLMQ";
String secretKey = "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";
AWSCredentials credentials = new BasicAWSCredentials(accessKey, secretKey);

// get the SQS service
AmazonSQS sqs = new AmazonSQSClient(credentials);

List<String> attributes = new ArrayList<String>();
attributes.add("All");

// receive messages from SQS
ReceiveMessageRequest request = new ReceiveMessageRequest(
    "https://queue.amazonaws.com/205414005158/queue1");
request.setAttributeNames(attributes);

ReceiveMessageResult result = sqs.receiveMessage(request);
List<Message> messages = result.getMessages();

for (Message message : messages) {

  System.out.println("receiving message " + message.getMessageId());

  Date pickedUpTime = new Date();

  // process the message, and delete it from the queue...
  boolean succeeded = process(message);

  sqsLogger.logMessage(message, pickedUpTime, succeeded);

}
```


www.it-ebooks.info

Queue size

We can't read the queue size from our SimpleDB domain, but that is not necessary. We can get the queue attributes, indicating the state and containing two interesting attributes for the purpose of understanding the size of the queue:

`ApproximateNumberOfMessages`
: This number tells us how many messages are currently in the queue waiting to be received and deleted. It reports on visible messages only, showing part of what the queue contains.

`ApproximateNumberOfMessagesNotVisible`
: This shows how many messages have actually been received but not yet deleted. You can regard these as messages in progress if you use the receive/delete sequence as AWS intended.

Combining these two gives us the approximate size of the queue. If you want to see trending of the queue size, check these values during processing and add them to the SimpleDB log. Doing this with every message incurs quite a significant overhead, however. We are not really interested in queue size outside of calamity alerting; *latency* and *throughput* (described in the next section) are sufficient for our needs.

In the following example, we are retrieving the attributes of one specific queue. If you want to monitor all the queues, you can first make a call to `ListQueues` and then check the size for each:

```
// get the SQS service
AmazonSQS sqsService = ...

// get all the attributes of the queue
List<String> attributeNames = new ArrayList<String>();
attributeNames.add("All");

// list the attributes of the queue we are interested in
GetQueueAttributesRequest request =
  new GetQueueAttributesRequest("https://queue.amazonaws.com/205414005158/queue1");
request.setAttributeNames(attributeNames);

Map<String, String> attributes =
  sqsService.getQueueAttributes(request).getAttributes();

int messages = Integer.parseInt(attributes.get("ApproximateNumberOfMessages"));
int messagesNotVisible = Integer.parseInt(
    attributes.get("ApproximateNumberOfMessagesNotVisible"));

System.out.println("Messages in the queue: " + messages);
System.out.println("Messages not visible: " + messagesNotVisible);
System.out.println("Total messages in the queue: " + messages + messagesNotVisible);
```


www.it-ebooks.info

Latency

For our customer Marvia, we set up queues with a strict priority policy and specific guarantees of quality of service, and therefore monitoring the latency is particularly important. Remember that we implemented two queues with different priorities in Chapter 4. For one queue, we promise service that is as fast as possible, which we want to be within 5 minutes. The other queue is "when it's your turn," but should be within 15 minutes on average.

We will use these measures to tune our render farm. Messages should be picked up reasonably fast to allow time for processing, meaning instances should be readily available. For this to happen, we should make sure there's the right number of instances running.

And processing the 15-minute queue should be just fast enough. Looking at the latency will help us determine the optimum triggers for autoscaling our farm of PDF renderers.

We can't get the latency with one SimpleDB query. We have to get a range and calculate the average, min, and max values by hand. We are not dealing with huge sets of numbers, and we intend to query sparsely, say, once an hour at most. This is how we query SimpleDB and calculate latency in the `SQSLogger` class we created:

```
class SQSLogger {

// ...

  /**
   * Get the average latency of messages served from the given start datetime
   * to the given end datetime.
   * Return value is a long expressed in milliseconds
   */
  public long getLatency(Date start, Date end) throws ParseException {

    long count = 0;
    long totalLatency = 0;
    String nextToken = null;

    // retrieve all the items which are in the
    // date range we want
    SelectRequest request = new SelectRequest(
      "select SentTimestamp, PickedUpTimestamp, MessageBody " +
      "from sqs_log " +
      "where SentTimestamp > '"+ format.format(start) +"' " +
      "and PickedUpTimestamp < '" + format.format(end) + "'");

    do {
      request.setNextToken(nextToken);
      SelectResult result = simpleDB.select(request);
      nextToken = result.getNextToken();

      for (Item item : result.getItems()) {

        String sentTimestamp = null;
```


www.it-ebooks.info

```
        String pickedUpTimestamp = null;

        for (Attribute attribute : item.getAttributes()) {

          if ("SentTimestamp".equals(attribute.getName())) {
            sentTimestamp = attribute.getValue();

          } else if ("PickedUpTimestamp".equals(attribute.getName())) {

            // we need the earliest PickedUpTimestamp
            if (pickedUpTimestamp == null ||
                pickedUpTimestamp.compareTo(attribute.getValue()) > 0) {
              pickedUpTimestamp = attribute.getValue();
            }

          }
        }

        totalLatency +=
          format.parse(pickedUpTimestamp).getTime()
            - format.parse(sentTimestamp).getTime();
        count++;
      }
    } while (nextToken != null);

    // return the average
    return (count != 0 ? totalLatency / count : 0);
  }

  /**
   * Helper method which returns the latency of the past given period of time,
   * in milliseconds.
   */
  public long getLatency(int seconds) throws ParseException {
    Date now = new Date();
    Calendar before = Calendar.getInstance();
    before.setTime(now);
    before.add(Calendar.SECOND, - seconds);
    return getLatency(before.getTime(), now);
  }


// ...


}
```

Using the helper method, we can calculate latency in the past hour, day, week, or month, for example:

```
System.out.println("Latency (in minutes) for the last hour is " +
    sqsLogger.getLatency(60 * 60) / 1000.0f / 60);

System.out.println("Latency (in minutes) for the last day is " +
    sqsLogger.getLatency(24 * 60 * 60) / 1000.0f / 60);
```


www.it-ebooks.info

```
System.out.println("Latency (in minutes) for the last week is " +
    sqsLogger.getLatency(7 * 24 * 60 * 60) / 1000.0f / 60);

System.out.println("Latency (in minutes) for the last month is " +
    sqsLogger.getLatency(30 * 24 * 60 * 60) / 1000.0f / 60);
```

This is a generic way of calculating the latency of an SQS queue. For Marvia, we can log the message metadata in separate domains to be able to differentiate the latency for each queue.

Throughput

Throughput is not just a vanity metric you can use to tweet about the immense numbers of jobs per second you handle—it actually helps to gain insight into potential problems. Because the app gets more and more complex, measuring the throughput of queues gives a high-level overview of the performance of the entire system. And as with most of our other measurements, we need it to test assumptions. Growth in the "orders" queue should show a related growth in the "ready for shipping" queue, if that is what is expected.

With our SimpleDB logs, we can easily calculate throughput over a range by doing a simple `select`. We are interested in both alerting on certain thresholds as historical trending and comparison:

```
public class SQSLogger {

  // ...

  /**
   * Returns the number of messages served from the given start timestamp to the
   * end timestamp.
   */
  public long getThroughput(Date start, Date end) {

    SelectRequest request = new SelectRequest(
        "select count(*) " +
        "from sqs_log " +
        "where SentTimestamp > '"+ format.format(start) +"' " +
        "and ProcessedTimestamp < '" + format.format(end) + "'");

    SelectResult result = simpleDB.select(request);

    for (Attribute attribute : result.getItems().get(0).getAttributes()) {
      if ("Count".equals(attribute.getName())) {
        return Long.parseLong(attribute.getValue());
      }
    }
    return 0;
  }


  /**
   * Helper method which
```


www.it-ebooks.info

```
     * returns the number of messages served in the past given period of time.
     */
    public long getThroughput(int seconds) {
      Date now = new Date();
      Calendar before = Calendar.getInstance();
      before.setTime(now);
      before.add(Calendar.SECOND, - seconds);
      return getThroughput(before.getTime(), now);
    }

  }
```

In the same way that we showed the latency, we can show the throughput:

```
System.out.println("Throughput in the last hour is " +
    sqsLogger.getThroughput(60 * 60));

System.out.println("Throughput in the last day is " +
    sqsLogger.getThroughput(24 * 60 * 60));

System.out.println("Throughput in the last week is " +
    sqsLogger.getThroughput(7 * 24 * 60 * 60));

System.out.println("Throughput in the last month is " +
    sqsLogger.getThroughput(30 * 24 * 60 * 60));
```

Again, by logging the messages in different SimpleDB domains, we can calculate the throughput for the two Marvia queues separately.

SimpleDB

SimpleDB is a relatively static data store in most applications. Not that it never changes, but the items stored are kept for a longer time than when they flow through SQS or SNS. The limits imposed on SimpleDB domains are strict, and we want to know when we are nearing these boundaries. But apart from the size (in different dimensions), we also want to know something else.

Because SimpleDB is schemaless, you can add and remove attributes at will, per item. This is convenient, but for domains that stay around longer, the risk of fragmentation arises. Because we have different groups of people building different parts of our applications, there should be some form of contract. SimpleDB does not enforce that contract, and some breaches might go unnoticed.

Size

The maximum size of a domain is one billion attributes, or 10 GB of storage. You can retrieve the list of attributes directly from the domain metadata, but the storage is split into several parts. We can look independently at the total size (in number of items or bytes) of items, attribute values, and attribute names. Adding the bytes version of these measures will get you something that is close to the total size of the domain.

www.it-ebooks.info

We would like to treat our domains as disks (volumes) and be notified when we exceed a certain percentage of our available storage space. Reading out the different metadata elements from a domain in Java looks like the following—we'll leave it up to you to create a secure URL you can regularly check with Monit, cron, or some other monitoring utility:

```
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.model.DomainMetadataRequest;
import com.amazonaws.services.simpledb.model.DomainMetadataResult;


// get the SimpleDB service
AmazonSimpleDB simpleDB = ...

// get the metadata for a domain called "my_domain"
DomainMetadataResult result = simpleDB.domainMetadata(
    new DomainMetadataRequest("my_domain"));

// number of attributes
// check if we already have 90% of the maximum number of attributes allowed
if (result.getAttributeNameCount() > 900000000) {
  // ...in that case, send a warning to myself...
}

// calculate the approximate size of the domain in bytes
// check if we already use 80% of the maximum allowed size of a domain
if (result.getAttributeNamesSizeBytes() +
    result.getAttributeValuesSizeBytes() +
    result.getItemNamesSizeBytes() > 10 * 1024 * 1024 * 1024 * 0.8) {
  // ...in that case, send a warning to myself...
}
```

Fragmentation

Even though SimpleDB allows you to have different attributes for items in the same domain, normally you expect that all the items from a domain are of a certain type. This means they have more or less the same attributes. If this isn't the case and each item is different, we say the domain is *fragmented*. Figure 7-1 shows a sample domain of users for Marvia that is very fragmented. Figure 7-2 shows the same domain with no fragmentation.

Status | Parameters | Result1

	itemName()	username	name	uses_API	PDFs_requested	fair_program	contract_expiration	company
1	000004	locomundo	Flavia Paganelli	yes				
2	000003	truthtrap	Jurg van Vliet	yes				
3	000001	telegraaf			30	no	2011-04-30	
4	000002	cineville			10	yes	2011-12-31	
5	000005	johnny	John Perez	yes				
6	000006	mary	Mary Smith					Casa e Luce

Figure 7-1. Users SimpleDB domain with fragmentation

www.it-ebooks.info

	Status	Parameters	Result1					
	itemName()	username	name	uses_API	PDFs_requested	fair_program	contract_expiration	company
1	000004	locomundo	Flavia Paganelli	yes	1	yes	2011-12-31	9Apps
2	000003	truthtrap	Jurg van Vliet	yes	4	yes	2011-12-31	9Apps
3	000001	telegraaf	Bas van Dieren	yes	30	no	2011-04-30	Telegraaf
4	000002	cineville	Laura Gonzalez	yes	10	yes	2011-12-31	Cineville
5	000005	johnny	John Perez	yes	0	no	2011-04-30	Telegraaf
6	000006	mary	Mary Smith	no	2	yes	2011-06-30	Casa e Luce

Figure 7-2. Users SimpleDB domain with no fragmentation

Items can have at most 256 key/value pairs. So if your domain has more than 256 different attributes, there must be some form of fragmentation: there are items that don't share some attributes. For example, if we have two items and 512 different attributes, the fragmentation is huge, because the two items do not have any attributes in common.

From the perspective of an outside mediator, we would like to have a sense of the fragmentation of items, because if there is a lot of fragmentation, there are more likely to be errors in the applications that access SimpleDB. When programming, it is more difficult to know what to expect from your dataset if each item is different. In relational databases, we can resolve this issue with schemas. The freedom that SimpleDB gives can be dangerous if we are not careful.

It is difficult to calculate fragmentation precisely because we can't easily iterate over millions of items, and the available operations in SimpleDB are limited. We can, however, use the domain metadata to get an idea. `ItemCount` tells us how many items we have, and `AttributeValueCount` indicates the number of name/value pairs in the domain. With total disregard for distribution, we can calculate the average number of name/value pairs per item, getting an approximation of how many attributes an item has. We are assuming there aren't a lot of values that correspond to multivalued attributes.

We can also get the number of unique attribute names in the domain with `AttributeNameCount`. If we have no fragmentation, the average number of attributes is equal to the number of unique attribute names. Again, this is true when the number of values for multi-valued attributes is not significant. The following calculations are equivalent:

Fragmentation = 1 / ((AttributeValueCount / ItemCount) / AttributeNameCount)

Fragmentation = (ItemCount / AttributeValueCount) * AttributeNameCount

For the example shown in Figure 7-1, this formula gives (6 / 20) * 7 = 2.1. In the non-fragmented example shown in Figure 7-2, we have (6 / 42) * 7 = 1. No fragmentation.

With this equation, if you add new attributes (higher `AttributeNameCount`) and you don't add those to all your items (which would increase `AttributeValueCount ItemCount` times), the fragmentation will increase. For example, consider a clean domain of users consisting of 12 attributes. Introducing 4 new attributes will change the fragmentation from 1 to about 4/3 for a sufficiently large domain.

www.it-ebooks.info

Because your app is now subject to access by more than one team, it is only logical that changes are introduced that will affect others. Having this informer in our arsenal of techniques lets you detect the changes in the air quickly and help your teams coordinate to resolve them. And now, finally, we can quote a Greek philosopher. It was Heraclitus who said: "Πάντα ῥεῖ καὶ οὐδὲν μένει" or "Everything flows, nothing stands still." We often hear this as "Change is the only constant," which becomes more and more apparent as systems become more complex.

RDBMSs tackle this problem by making change difficult, forcing you to do migrations. SimpleDB makes it much easier by allowing you to upgrade items at your leisure. But this strength is also its weakness, and we watch this constant closely.

SNS

As discussed before, SQS is great as a funnel, allowing as many (uncontrolled) writers and as many readers as you want. In this way, SQS buffers write bursts. SNS, on the other hand, is great for broadcasting information. It doesn't buffer, it creates bursts. At this moment, the possibilities for sending messages are not ready to expose to end users, as there is no way of determining the whole content and format of your email messages. You will probably want to process the message and send it through. But because SNS does not limit the number of subscribers per topic, you might get yourself into trouble.

Burst factor

And that is basically the only thing we are interested in: the *burst factor* of our SNS topics. This is the number of subscribers to a topic. If you use SNS to administer subscribers, as we chose to do in our examples in Chapter 4, we might create bursts we are not ready for. How many bursts you can handle in your app depends on many factors, of course. To get the number of subscriptions per topic, you can call the `ListSubscriptionsByTopic` action, using, as always, `nextToken` to get all the pages and count them, as shown here:

```
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.sns.AmazonSNS;
import com.amazonaws.services.sns.AmazonSNSClient;
import com.amazonaws.services.sns.model.ListSubscriptionsByTopicRequest;
import com.amazonaws.services.sns.model.ListSubscriptionsByTopicResult;
import com.amazonaws.services.sns.model.Subscription;


// get the SNS service
AmazonSNS sns = new AmazonSNSClient(new BasicAWSCredentials(
    "AKIAIGKECZXA7AEIJLMQ", "w2Y3dx82vcY1YSKbJY51GmfFQn3705ftW4uSBrHn"));

String nextToken = null;
int subscriptions = 0;

do {
```


www.it-ebooks.info

```
    // call service ListSubscriptionsByTopic
    ListSubscriptionsByTopicResult result =
      sns.listSubscriptionsByTopic(
          new ListSubscriptionsByTopicRequest(
          "arn:aws:sns:us-east-1:205414005158:decaf-alarm")
          .withNextToken(nextToken));
    nextToken = result.getNextToken();

    // show the subscriptions
    for (Subscription subscription : result.getSubscriptions()) {
      subscriptions++;
      System.out.println("Subscription: " + subscription);
    }

  // repeat until there are no more pages
  } while (nextToken != null);

  System.out.println("There are " + subscriptions + " subscriptions for this topic");
```

Which will show something like this:

```
Subscription:
  {SubscriptionArn: arn:aws:sns:us-east-1:205414005158:decaf-alarm:…,
  Owner: 205414005158, Protocol: email, Endpoint: flavia@9apps.net,
  TopicArn: arn:aws:sns:us-east-1:205414005158:decaf-alarm, }
Subscription:
  {SubscriptionArn: arn:aws:sns:us-east-1:205414005158:decaf-alarm:…,
  Owner: 205414005158, Protocol: email, Endpoint: jurg@9apps.net,
  TopicArn: arn:aws:sns:us-east-1:205414005158:decaf-alarm, }

There are 2 subscriptions for this topic
```

Understand

When your app gets very big, everything slows down a little bit. It's like a river: near its source, it can appear to be anything from a trickle to a raging stream. Further downstream, it gets bigger and bigger, but looks much calmer unless you limit the space it has to flow through. Only then do you see the real force the river is generating. But give it more space than it needs, and you will have a lake.

Your app is like that—a big river with stuff going through. Most of the time, it runs fine, but you need to understand where the weak spots are—where you will have droughts and where you will have overflows.

You can find remedies inside the individual components with techniques like Auto Scaling and ELB, as seen in Chapter 6. You can also look outside of the individual components—SQS, SimpleDB, and SNS services—to understand what's going on.

www.it-ebooks.info

Imbalances

Imbalances are subtle—nothing breaks, but something is wrong. For example, if your processing capacity of queue items cannot keep up during busy periods, the queue latency might be too high. This imbalance is not easy to detect inside of the individual components. The problem might not be that your EC2 instance is too slow to process queue items, but that you need more instances doing the job.

In the case of Marvia, for example, not having enough capacity to deal with our job queues cannot be remedied with Auto Scaling. You must always overprovision it to be ready to process requests and guarantee quality of service. This can only be done manually. Luckily, the queue handles the peaks, so these machines have time to keep churning out PDFs. But if generating PDFs begins to take more and more time, customers will become uncomfortable with our app and wonder what is going on. And that is something we should prevent from happening at all times.

Bursts

There is nothing subtle about bursts. You will definitely know something is wrong, because things will break. You can't prevent bursts from breaking things occasionally, but the system will break only one component, while the app itself will still be mostly OK. This is not ideal, but customers tend to understand these kinds of things. They instinctively realize systems have breaking parts, like a light on a car. But if the entire car doesn't work anymore, they can get irritated.

While we can't prevent bursts, we can predict them. Predicting bursts means learning where the risks are and planning for the risks we want to cover. For example, take Kulitzer, where we use SNS to propagate notifications to users. We assume that most of the time, a contest will not have more than 50 users, and sending a notification to 50 users at the same time doesn't make our app sweat. (Remember, we send the notifications through our own web instances because we need to send something decently formatted and in the right design.)

We also know that the size of a group inside such a community will probably be subject to a power law, like Pareto's 80/20 rule, which says that 20% of the groups have 80% of the users. And because of this, it is inevitable that at some point in time we'll get contests with thousands or even tens of thousands of users. It will not happen often, but it will happen.

Improvement Strategies

SQS, SimpleDB, and SNS are great services for decoupling your system. They allow you to scale your application, and they also allow you to scale your organization. The essence of a queue is that it buffers, making it suitable for asynchronous processing. SimpleDB captures information and allows you to do basic querying over substantial

www.it-ebooks.info

quantities of data very quickly. And SNS accelerates the flow of information by distributing a message to all the subscribers of a particular topic.

You can configure SimpleDB a bit like a disk, in that you can configure it to notify you when it is nearly full so you can either tidy up or buy yourself another disk. Consider queues carefully, because they can introduce subtle degradation of your services that are not easily detected on either side of the queue. And SNS is potentially hazardous because it can create bursts of information.

Luckily, you can use the underlying patterns (and in some cases, the same AWS services) to neutralize these events.

Queues Neutralize Bursts

Queues are like the lakes on a river; they act like buffers. You can use SQS to buffer the bursts. In the case of Kulitzer, we can burst jobs to a queue, where we can process them a bit slower. This is not an ideal situation because we are dealing with winner announcements of potentially media-worthy contests, and you can't have one group of people being notified significantly sooner or later than another group.

We can look into another infrastructure component with queue characteristics: a mail transfer agent (mail server). We can set up a mail farm (with all the techniques detailed in Chapter 3) that is capable of handling/processing bursts of up to 10,000 mail messages within a minute, for example. This approach has two advantages:

- We know what traffic flow we have and that bigger bursts are buffered.
- We can scale this approach easily.

In this particular example, we do use a queue, but not SQS. The principle remains the same: a queue neutralizes bursts by buffering input.

Notifications Accelerate

The property of SNS we are trying to prevent from causing too much harm can also be very useful. A queue doesn't do much by itself—there have to be processes polling it to get some sort of action. Building daemons that scale themselves is a bit difficult, and you don't always want hundreds of them polling the SQS queue. You can use SNS to wake up more daemons instead of waking them up from within.

In this way, you can control the growth rate of your processing power by adding or deleting subscribers to the topic without tweaking the component itself. You can combine this technique with Auto Scaling, for example. It is suitable for queues with big variations in throughput. You want to have just enough daemons to handle the queue as it is—no more, no less.

www.it-ebooks.info

In Short

Once you have your big, scalable system using AWS services, you still need to observe it to make sure it is running smoothly. In this chapter, we looked at several properties of AWS components that can help you measure its performance.

For SQS, we looked at queue size, latency, and throughput. You will want to keep the size of your queues small, as well as the latency; these give you an idea of how fast the system is processing jobs, for example. The throughput can give you an overview of the performance of your system.

For SimpleDB, we looked at size and fragmentation. We care about the size of SimpleDB domains because there are limitations set by Amazon, and if we are reaching those limits we have to take action (e.g., by dividing your items into more domains). Fragmentation is about the structure of your dataset, and we are interested in it to avoid failures of application components that use SimpleDB.

For SNS, we looked at the burst factor, which is the number of subscriptions you have per topic.

www.it-ebooks.info

CHAPTER 8
And Now…

We believe that every app is unique and requires special attention and care. You spend considerable energy and other resources developing products and services powered by web applications. AWS makes it easier to transform that energy into value for your users.

The intention of this book is to get you started with AWS, and more importantly to keep you going when success arrives. We have explained how virtual infrastructures differ from physical infrastructures. It's easy (and cheap) to experiment, and it's affordable to scale. With services like SQS, SimpleDB, and SNS, you can take everything a step further—you are ready for what they call "Internet-scale."

We have documented our experience in building scalable applications and helping developers and development teams cope with their success. Success should be a joy, but everyone will encounter hiccups accompanied by a fair share of stress. But we believe that in between these periods, you should be happy and proud of what you've accomplished with the AWS services at your disposal.

Other Approaches

AWS is an Infrastructure as a Service-type cloud. The core elements are similar to what you are used to with your virtual servers. On top of that, AWS adds a layer of services to help you deal with scaling. Another public cloud that is comparable to AWS is Windows Azure.

Google App Engine (GAE) is also a public cloud, but has an entirely different approach than AWS. GAE offers a full-service computing environment for your application and provides transparent scalability. You don't really have infrastructural components like an instance or a load balancer—everything is taken care of by the cloud. Not everything is suited for this approach, but you can build beautiful applications on GAE. It is usually classified as Platform as a Service (PaaS).

www.it-ebooks.info

Heroku is another kind of PaaS, specifically tailored to Ruby (Rails) applications. Its creators call it a platform, and built it entirely on AWS. We think it is best compared to GAE. At the time of this writing, Heroku has hit the 100,000 apps mark. This shows how powerful this approach is—it enables developers to quickly and easily deploy their work.

Private/Hybrid Clouds

Sometimes the "public cloud people" try to make you believe their approach is the only viable one. However, there are lots of valid reasons for doing it yourself. The essence of a public cloud is to limit waste. We think the most wasteful thing you can do is to get rid of your own servers before they expire.

If you want to utilize the cloud paradigm on your own premises, you can do so. On top of virtualization technology like VMWare, there are some software stacks to expose your physical infrastructure as a cloud. We would like to mention one solution called Eucalyptus. This startup, led by Marten Mickos of MySQL fame, aims to provide the technology to transform your infrastructure into your own private AWS. By exposing this cloud through AWS APIs, it's as if you're working with a public cloud, making it easier to use both AWS and Eucalyptus and get the most out of your applications.

Thank You

Writing this book has been an adventure, and we have learned many things in the process. We sincerely hope that we have given you the tools to start building your app on AWS. By showing you many of the problems we faced and solutions we came up with together with our partners, we tried to give you a head start in designing, building, and operating your own apps.

The cloud is here to stay. We have barely begun to grasp the possibilities this way of handling computing resources enables. AWS is relentlessly innovating, and we don't think it plans to take it easy anytime soon. While writing this book, many new features and functionalities were introduced. We managed to include most of them, but AWS development and introduction of new features are always in progress.

Thank you for picking up a copy of this book. Just as AWS will continue to innovate, we'll continue to build our apps on top of it. Good luck with building yours!

www.it-ebooks.info

Index

Numbers

9Apps, xiii

A

access keys, 17
Agile Manifesto, 7
alerts, 134–136
Alestic, 24
Amazon Web Services (see AWS), xiii
AMIs, 23
AssetPackager, 44
attachment_fu, 54
attributes, 85
Auto Scaling, 6, 10, 60–66
  alarms and policies, 62
    changing alarms, 65
  autoscaling groups, 61
    changing or decommissioning, 66
  in production, 64
  launch configurations, 61
    replacing, 65
  pausing, 64
  semiautoscaling, 64
  setting up, 60
  tuning, 138
AutoScalingGroupName, 130
availability zones, 7, 21
AvailabilityZone, 131
AWS (Amazon Web Services), xiii, 1–14, 157
  account set-up, 16
  cost, 1, 17
  environment, setting up, 16
  history, 1–14
    AWS developer responsibility, 11
    decoupling, 2
    early versions, 1
    EC2, 4
    infinite storage, 3
    iterative infrastructure engineering, 7
    scalable data storage, 5
  scaling, 1
  services in order of development, 2
AWS credentials, 17
AWS Management Console, 19
AWS PHP Library installation, 78
AWS Policy Generator, 19
AWS SDKs for Android and Java, 82
AWS Toolkit for Eclipse, 96

B

backups of volumes, 45–49
  backup script, 46
  cron for making and deleting backups, 48
  SimpleDB client installation, 46
Barr, Jeff, 3
Bezos, Jeff, 1
BinLogDiskUsage, 132
Bj plug-in, 54
buckets, 41
burst factor, 152
burst neutralization, 155

C

Canonical AMIs, 24
CDN (content delivery network), 6
Cineville.nl, 77
client-based monitoring, 114
cloud computing, 4

We’d like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

www.it-ebooks.info

CloudFront, 6, 21, 41
    distributions, 9
    setting up, 41
CloudWatch, 6, 118
    EC2 CloudWatch metrics, 130
    ELB CloudWatch metrics, 121, 131
    graphs, interpreting, 122
    RDS CloudWatch metrics, 132
    region visualization, 121
command-line tools, 17, 54
component based software engineering, 2
conditional put/delete, 86
config.action_controller.asset_host, 43
Configure Firewall screen, 27
consistent read, 86
CPUUtilization, 130
custom images, 31

D

DatabaseClass, 132
DatabaseConnections, 133
databases, 53
    (see also RDS; MySQL)
    launching, 9
    scaling, 53
DB Instance Wizard, 36
DB parameter groups, 39
DB security groups, 37
DBInstanceIdentifier, 132
Decaf, xiii, 20
    monitoring queues with SQS, 81
    SimpleDB, usage in, 95–99
Decaf Monitor, 114
decoupled systems, managing, 141, 153
    bursts, 154
    imbalances, 154
    improvement strategies, 154
        burst neutralization, 155
decoupling, 2
dimensions, 120, 129
    command line, using from, 133
Directness, xiii, 114
DiskReadBytes, 131
DiskReadOps, 130
DiskWriteBytes, 131
DiskWriteOps, 131
distributions, 41
domains, 85
downtime, using to advantage (see monitoring), 113
Dynamo, 5

E

EB (Elastic Beanstalk), 70–72
EBS (Elastic Block Store) volumes, 8, 28
    creating and using, 30
EBS-backed AMIs, 24
EC2 (Elastic Computer Cloud), 4
    API tools, 17
    dimensions and metrics, 130
    instances, 8
ec2-create-image command, 31
ec2-describe-regions command, 18
EIPs (Elastic IPs), 29
    creating and associating, 31
Elastic Block Store volumes (see EBS), 8
Elastic Computer Cloud (see EC2), 4
elastic infrastructure, 51
Elastic IP addresses, 8
ELB (Elastic Load Balancing), 6, 10, 55–59
    CloudWatch metrics, 121
    creating an ELB, 56
    difficulties, 59
    dimensions and metrics, 131
    handling additional web protocols, 59
    handling HTTPS traffic, 58
    Layar example, 56
endpoint, 100
EngineName, 132
ephemeral storage, 24
Eucalyptus, 158
event model, 99
eventual consistency, 52, 86

F

fragmentation, 150
free usage tier, 17
FreeStorageSpace, 133

G

GAE (Google App Engine), 157

H

HealthyHostCount, 132
Heroku, 158

www.it-ebooks.info

Howard, Ara, 55

I

IAM (Identity and Access Management), 19
ImageId, 130
images, 8, 23
image_tag method, 55
infrastructures
  expectations based on size, 136
  improvement strategies, 138
  managing large versus small, 136
initaws, 18
instance limits, 26
instance types, 25
instance-store Root Device Type, 24
InstanceId, 130
instances, 5, 8, 24
  launching, 27
  provisioning at boot/launch, 32
  setting up, 28
  tagging or key-value pairs, 26
InstanceType, 130
items, 85
iterative infrastructure engineering, 7

K

key pairs, 17
  creating, 23
Kulitzer, xiv
  example application, 15
    AMI, 23
    architecture, 21
    Auto Scaling, 60
    environment, 16
    geographic location, regions, and availability zones, 21
    instance, launching, 26
    Rails server creation on EC2, 22
    RDS database, 35–41
    S3 and CloudFront, 41–45
    user uploads, 54
    volume backups, 45–49
    Web/Application server setup, 24–35
  offloading image processing with SQS, 74
  SimpleDB, storing users with, 88
  SNS, implementing contest rules with, 100–105

L

Latency, 132
Layar, xiv, 56
ListDomains, 96
ListQueues, 82
load balancing, 6, 55
LoadBalancerName, 131

M

Marvia, xiv, 77
  SimpleDB, sharing accounts and templates with, 91–95
  SNS, monitoring PDF processing status with, 105–107
message passing, 2
message queuing, 73
message size, 74
metrics, 120, 130
Mickos, Marten, 158
mod_rewrite command, 59
Monit, 115
monitoring, 20, 113
  client-based monitoring, 114
  comparing instances, 137
  distinguishing expected from unexpected behaviors, 122
  improvement strategies, 124
    benchmarking and tuning, 124
  loss of instances, 122
  predicting bottlenecks, 124
  server-based monitoring, 118
  spikes in CPU usage, assessing, 122
  up/down alerts, 114
Multi-Availability Zone deployment, 37
MySQL, 8
  InnoDB versus MyISAM, 36
  version issues with RDS database, 41

N

namespace, 120
NetworkIn and NetworkOut, 131
nonautoscaling components, planning, 138
nonfilesystem style data, storage, 53
NoSQL, 10

O

Olson, Rick, 54

www.it-ebooks.info

optimization, 6

P

PaaS (Platform as a Service), 157
PHP Library installation, 78
Pingdom, 114
private/hybrid clouds, 158
public clouds, 157
Publitas, xiv, 45

Q

Quadruple Extra Large instance, 25
queue latency, 142
queue throughput, 142
queues, 2, 73

R

RDS (Relational Database Server), 5, 6, 35–41
  creating an RDS instance, 36
  databases, command-line launching, 9
  DB parameter groups, 39
  dimensions and metrics, 132
  disk access speed, 69
  MySQL and, 8, 35
    RDS version peculiarities, 69
    version issues, 41
  RDS Command Line Toolkit, 17
  scaling a relational database, 66–70
    tips and tricks, 69
  scaling out, 68
    differences in storage engines, 69
  scaling storage, 70
  scaling up or down, 66
    DB instance classes, 67
  slow query logging, 70
ReadIOPS, 133
ReadLatency, 133
ReadThroughput, 133
reduced redundancy storage, 77
regions, 7, 21
  specifying, 32
Relational Database Server (see RDS), 5
ReplicaLag, 132
report on computer memory, 125
Request Instances Wizard, 26
RequestCount, 132
right_aws, 76

S

S3 (Simple Storage Service), 3
  bucket measurement, 142
  setting up, 41
S3-backed AMIs, 24
S3/CloudFront presentation tier, 41, 52
scalable data storage, 5
scalable infrastructure, 9
scaling out, 8, 51
  S3/CloudFront presentation tier, 52
  tools, installing, 54
security groups, 26
Seibel, Joerg, 21
select, 97
servers, starting, 8
service oriented architecture (see SOA), 2
Simple Monthly Calculator, 17
Simple Storage Service (see S3), 3
SimpleDB, 5, 85
  client installation, 46
  Decaf, 95
    getting domain metadata, 98
    listing domains, 96
    listing items in a domain, 97
  decoupling, 87
  Kulitzer, storing users for, 88
    adding users, 90
    getting users, 91
  lack of joins, 85
  maintenance, 149
    handling fragmentation, 150
    size, 149
  Marvia, sharing accounts and templates, 91
    adding accounts, 92
    counter incrementation, 94
    getting accounts, 93
  relational databases, comparison with, 86
  snapshot administration, 45
  use cases, 87
SLA (service level agreement), 3
snapshots, 8
  deleting expired snapshots, 48
SNS (Simple Notification Service), 99–110
  Decaf, using in, 108–110
    listing topics, 109
    sending notifications on topics, 110
    subscribing to topics via email, 110

www.it-ebooks.info

Kulitzer, implementing contest rules for, 100–105
deleting topics, 104
preparing tools, 101
publishing messages to topics, 104
publishing to Facebook, 105
subscription to registration updates, 102
topics per contest, 102
management, 152
burst factor, 152
Marvia, monitoring PDF processing status for, 105
publishing and receiving status updates, 107
subscription and confirmation, 106
topics for accounts, creating, 105
notifications acceleration, 155
SQS, compared to, 99
SOA (service oriented architecture), 2
spot instance, 26, 35
SQS (Simple Queue Service), 2, 73–85
burst neutralization, 155
Decaf, monitoring queues in, 81
checking specific queue attributes, 84
getting queues, 82
reading queue attributes, 83
Kulitzer, offloading image processing for, 74
Marvia, priority PDF processing for, 77–81
QoS implementation, 78
reading messages, 80
writing messages, 79
performance, measuring, 142–149
queue latency, 146
queue size, 145
queue throughput, 148
SNS compared to, 99
SQS Scratchpad, 75
statistic, 121
sticky sessions, 10
storage, 3

U

UnHealthyHostCount, 132
unit, 121
up/down monitoring, 114
user data, 35

V

van Woensel, Arjan, 15
visibility timeout, 74
Vogels, Werner, 1, 11

W

where clause, 98
Windows Azure, 157
WriteIOPS, 133
WriteLatency, 133
WriteThroughput, 133

X

X.509 certificates, 17

www.it-ebooks.info