Survive your Success

Programming

Amazon EC2

O'REILLY®

Jurg van Vliet
& Flavia Paganelli

O'Reilly Ebooks—Your bookshelf on your devices!

When you buy an ebook through oreilly.com, you get lifetime access to the book, and whenever possible we provide it to you in four, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and Android .apk ebook—that you can use on the devices of your choice. Our ebook files are fully searchable and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at http://oreilly.com/ebooks/

You can also purchase O'Reilly ebooks through iTunes, the Android Marketplace, and Amazon.com.

Programming Amazon EC2
by Jurg van Vliet and Flavia Paganelli

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

Editors: Mike Loukides and Julie Steele
Production Editor: Adam Zaremba
Copyeditor: Amy Thomson
Proofreader: FIX ME!

Indexer:
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
February 2011: First Edition.

ISBN: 978-1-449-39368-7

[LSI]

1295635226

Table of Contents

CHAPTER 1

Introducing AWS

From 0 to AWS

By the late 1990s, Amazon had proven its success—it showed people were willing to shop online. Amazon generated $15.7 million sales in 1996, its first full fiscal year. Just 3 years later, Amazon saw $1.6 billion in sales and Jeff Bezos was chosen Person of the Year by Time magazine (*http://www.time.com/time/subscriber/personoftheyear/archive/stories/1999.html*). 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* (*http://oreilly.com/catalog/9780596005429*) 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 6 years of AWS evolution (refer to Figure 1-1 for a timeline). It is not just a historical journey, it is 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.

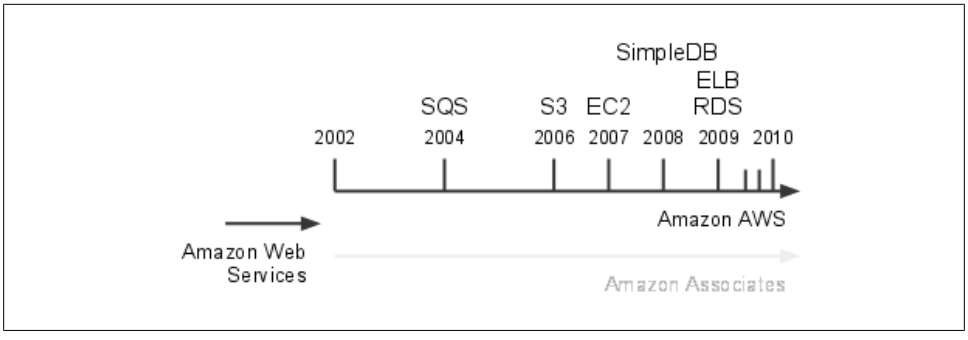

Figure 1-1. Timeline of AWS

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 problem they were designed to solve.

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 (*http://www.omg.org/*), and using the "component based software engineering" concept. Similar design principles were used to coin the more recent term SOA, or service oriented architecture (*http://en.wikipedia.org/wiki/Service-oriented_architecture*), 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 (*http://aws.amazon.com/sqs/*), 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 can have huge peaks and generate 80% of the orders in 20% of the time. You can have a component which processes

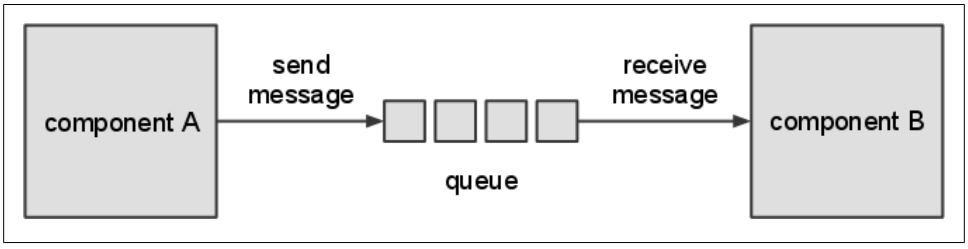

Figure 1-2. Passing messages using a queue

these orders, and a queue containing them. Your web application would put orders in the queue, and then your processing component can work on the orders the entire day without overloading your web application.

Infinite storage

In every application, storage is an issue. There is a very famous quote attributed to Bill Gates that 640k "ought to be enough for anybody." Of course, he denies having said this, but it does trigger some emotion. 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 (*http://aws.amazon.com/s3/*), or S3. It was released in 2006, two years after Amazon announced SQS. The time Amazon took to release it shows that storage was 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, Netflix must have 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" (*http://aws.amazon.com/about-aws/whats-new/2006/07/11/amazon-simple-storage-service-amazon-s3---continuing-suc*

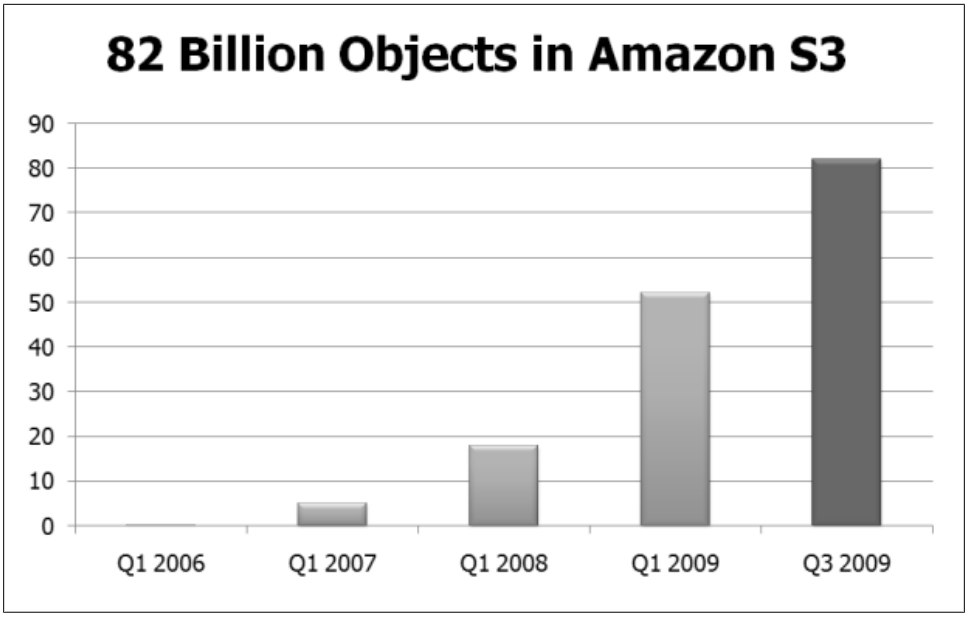

Figure 1-3. S3's huge popularity expressed in objects stored

cesses/). 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.

Computing Per Hour

Though we still think the most revolutionary of services is S3 because no one had solved the problem of unlimited store before, the service with the most impact is undoubtedly Amazon Elastic Compute Cloud (*http://aws.amazon.com/ec2/*), 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 didn't want each (small) team to just 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."

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 (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 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," (MySQL) 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 (*http://aws.amazon.com/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.

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 surely it will 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 (*http://aws.amazon.com/elasticloadbalancing/*), 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 ELB. To manage a group like this, AWS introduced Auto Scaling (*http://aws.amazon.com/autoscaling/*). 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.

Going Global

AWS first launched in the United States, on the east coast, in North 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 other. 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 (*http://en.wikipedia.org/wiki/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 a couple of servers out 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 it happened to us in the early days of Layar (*http://www.layar.com/*), 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 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—so called EC2 *instances*—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 does not bring your app down anymore.

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 applications 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. But 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 design for failure. You can assign Elastic IP addresses to an instance. If the instance dies or if 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 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 behaviour 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.

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 datastore, 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 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 only recent, but with that added, it 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 is practically infinitely scalable, but that comes at a cost. The management overhead of this scaling adds latency to the transactions. But, in the end, human labour 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. 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 scale easily, let alone automatically. The most expensive operation is the join. We already minimized the use of joins by caching objects, but it 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.

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 behaviour. 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 again. This pattern can be repeated again and again, of course.

"You Build It, You Run It"

Perhaps by chance, probably by design, AWS empowers development teams to become truly agile. They do 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 (*http://queue.acm.org/detail.cfm?id=1142065*), 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 he 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, but 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 ac-

tivity of operating applications can be surrounded with frustration and everyone feels powerless. Smaller companies often have a hosting provider, which can be very similar 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 provider.

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. And at the same time, the IT department has to make sure everything runs fine 24x7. It is in a continuous split between dealing with ultra long term and ultra short term; there seems to be nothing in between.

OK, the development team, then. 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 processes, 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 each 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 is 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.

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 just 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 3 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 amount 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 a certain moment, 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, we only want audio AND video." The project manager knows about the precondition and he also knows

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 last years of the 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.

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 (*http://www.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 it using Amazon RDS, a flavor of MySQL, because we love it. But you can just as well run your own PostgreSQL database, for example. It is a good idea to follow along,

and if you carefully stop your instances, it will definitely not 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 registation 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.

An AWS Account

Creating an AWS account is pretty straighforward. Go to the AWS website (*http://aws.amazon.com/*) 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).

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.

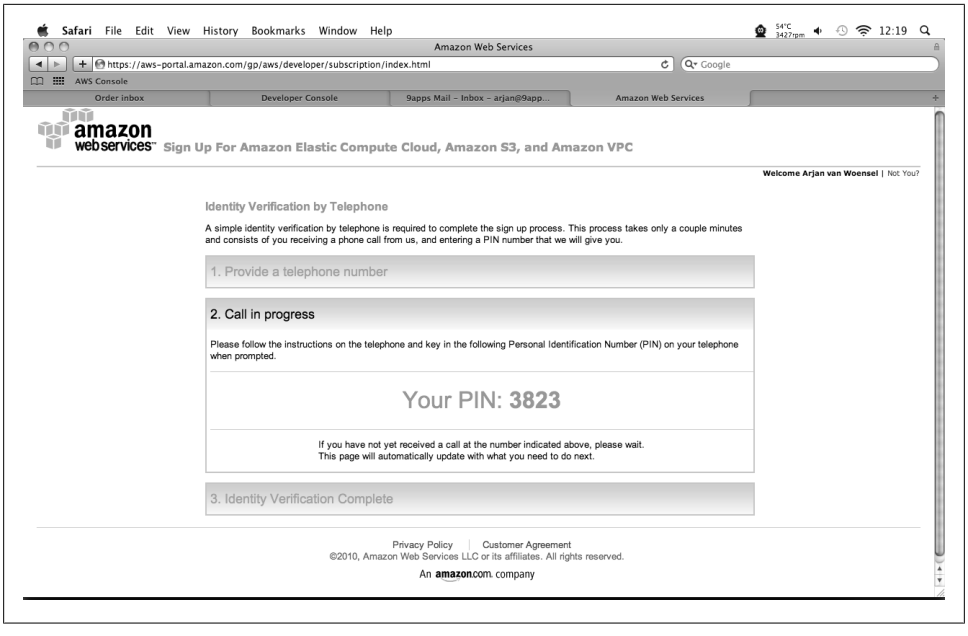

Figure 2-1. Identity verification

You might be wondering now... how much will all this this cost me? To start with, signing up for services doesn't cost anything. When you start actually using the services, Amazon provides a free usage tier (*http://aws.amazon.com/free/*) 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 5GB. 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 (*http://calculator.s3.amazonaws.com/calc5.html*). 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

Do take the few minutes it takes 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 them to script tasks that you want to automate.

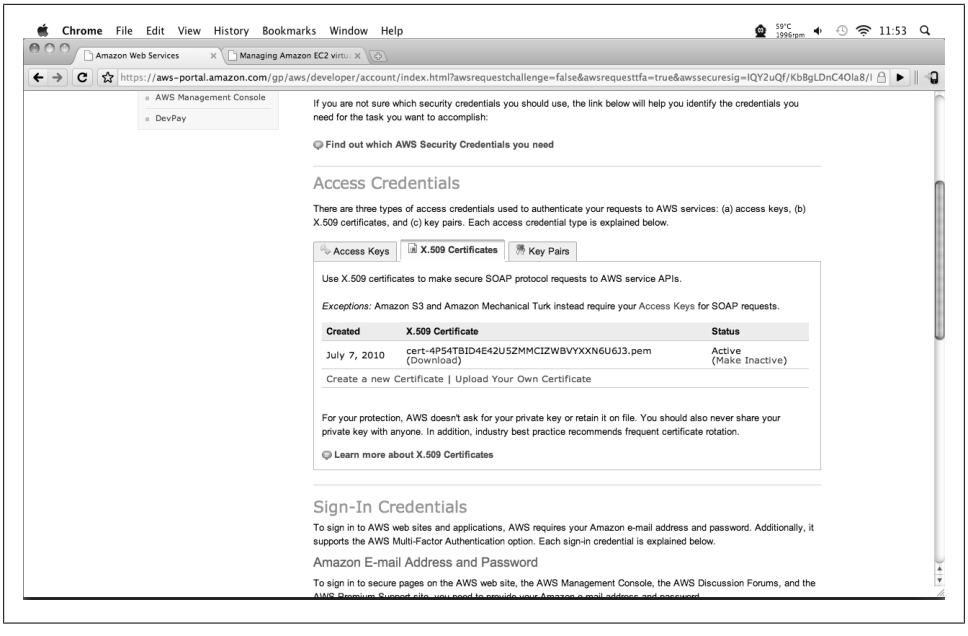

Figure 2-2. AWS credentials

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 flavours, for protecting CloudFront content and for accessing your EC2 instances

You will need X.509 credentials, the EC2 API tools (*http://developer.amazonwebservices.com/connect/entry.jspa?externalID=351*), and the RDS Command Line Toolkit (*http://developer.amazonwebservices.com/connect/entry.jspa?externalID=2928&categoryID=294*) 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).

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 value of the variables with the location 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
```

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.

In this book, we mostly use the credentials of the account itself. This simplifies working with AWS, but it does not comform to industrial-grade security practices. Amazon realized that and introduced yet another service, called Identity and Access Management (or 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
```

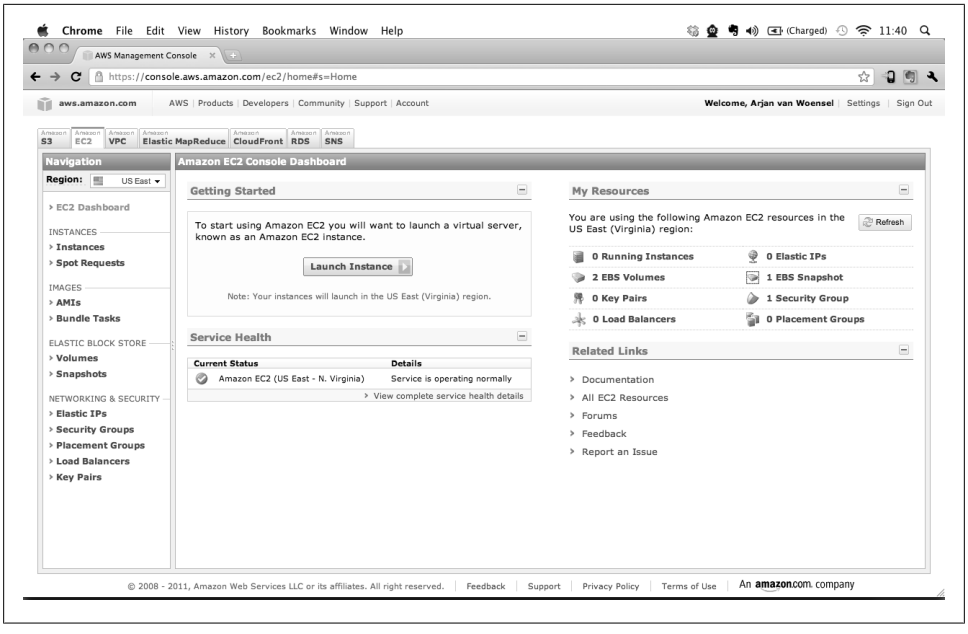

Figure 2-3. AWS Console

```
$ 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, other than mentioning it. To learn more, visit AWS Identity and Access Management (IAM) (*http://aws.amazon.com/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 (*http://awspolicygen.s3.amazonaws.com/policygen.html*). With this online tool, you can easily generate policies to be added to S3, EC2, or any of the other available AWS services.

AWS Console

What is there to say about the AWS 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 Management 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.

Other Tools

With the Amazon Web 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 get 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 how to operate your infrastructures.

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—Nothern 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 also 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 utilize this feature yet, 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.

Choose an Architecture

So, US East it is. What is next? Our application is standard, three-tiered; we have 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.

Our operating system of preference is Ubuntu, but you can run various other flavours 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 can you 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 order in which they are needed.

Create a Key Pair

Key pairs 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

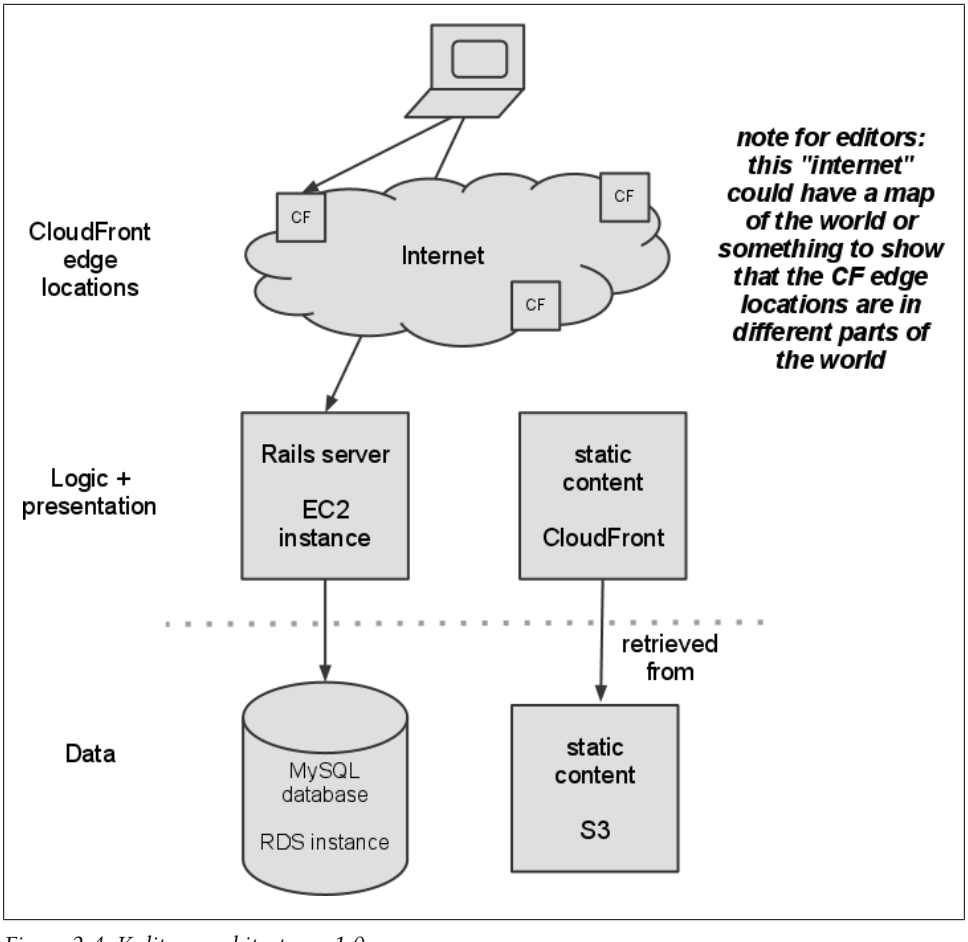

Figure 2-4. Kulitzer architecture v1.0

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 Amazon Web 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 flavia
```

where *flavia* will be the name of the key pair. You have to import your key pair to each region where you will use it.

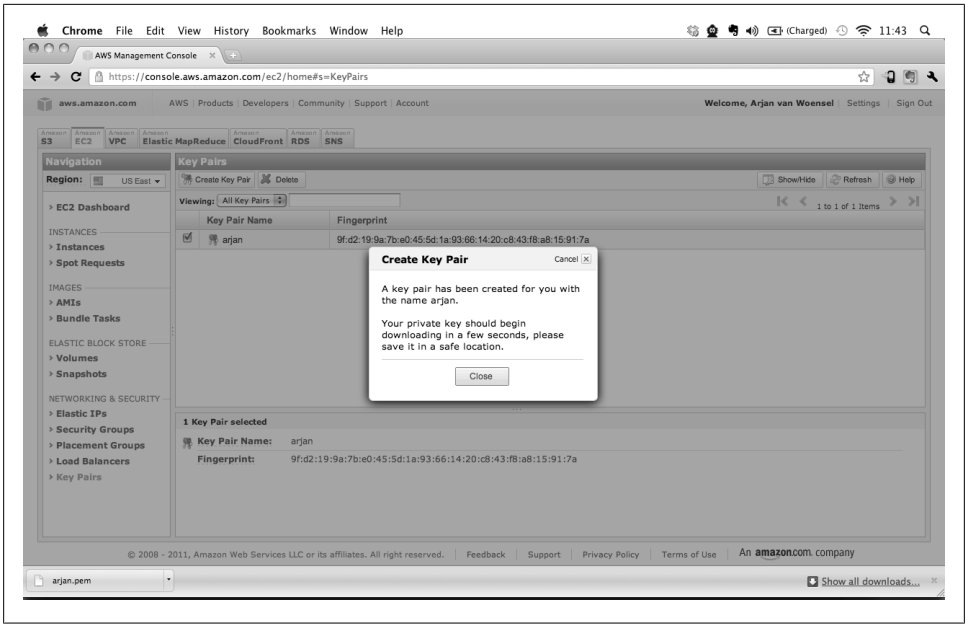

Figure 2-5. Create a key pair

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.

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 availabe AMIs, and you can create your own preconfigured one for your needs. The available operating systems include various flavours 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 (EBS will be described in detail later) 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-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 around, anywhere you want. This is problematic for EBS-backed AMIs. A situation where this poses a problem is 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 (*http://alestic.com/*). Though Ubuntu now builds and maintains its own EC2 AMIs (*http://cloud.ubuntu.com/ami/*), but we still find ourselves going to Alestic.

The web 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 is of type instance-store and the other one 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 they are 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 because it is launched from an immutable image. Instances come in types (*http://aws.amazon.com/ec2/instance-types/*). You can think of a type as the size of 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

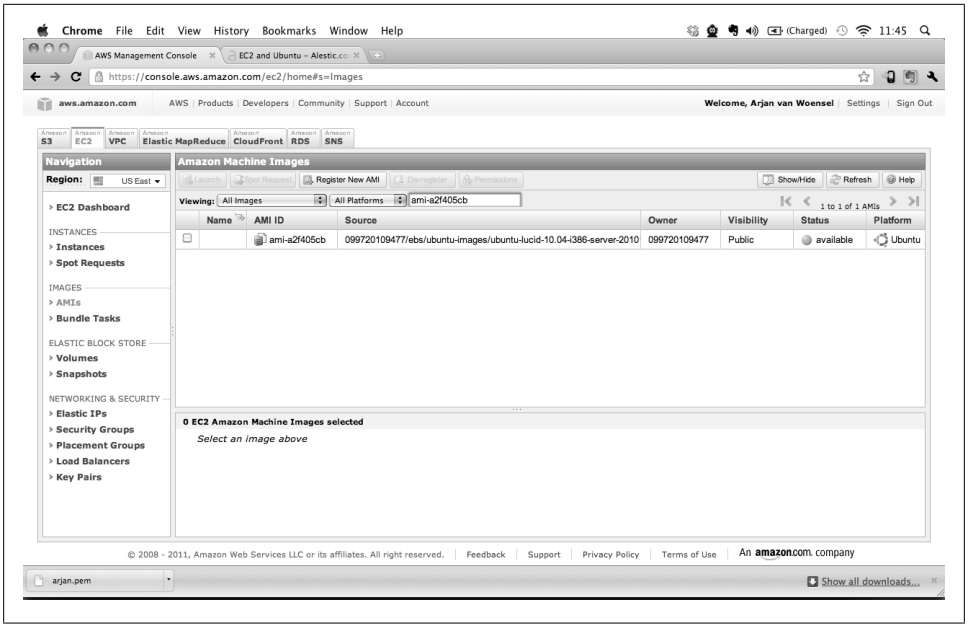

Figure 2-6. Find your AMIs

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

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."

Launch 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 availibility 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 (*http://aws.amazon.com/contact-us/ec2-request/*).

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 defaut, 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 it, 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 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

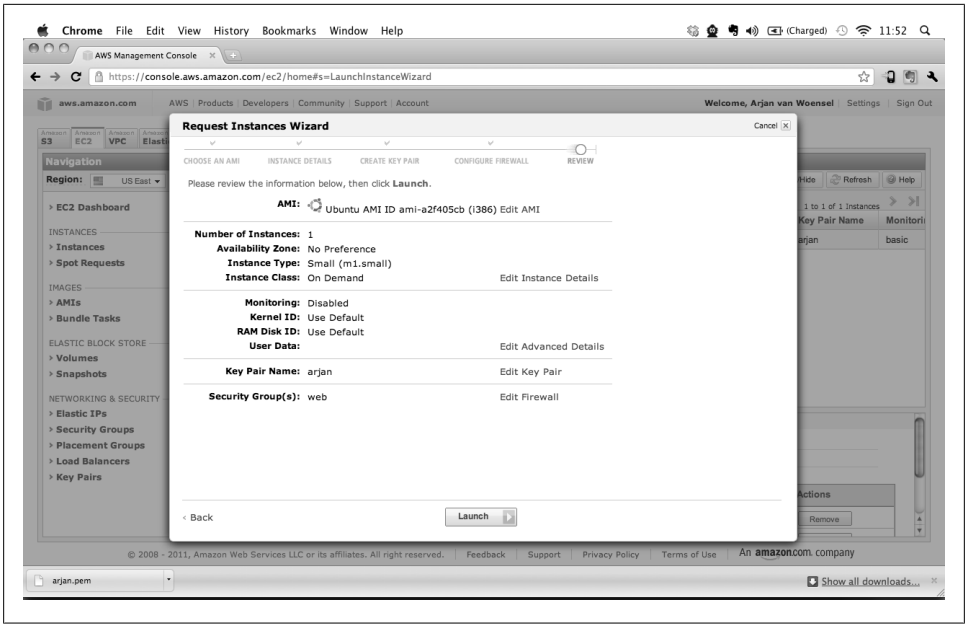

Figure 2-7. Ready to launch an instance

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. It would not have been so informative, but next time, you can do exactly the same 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 CIDR (Classless Inter-Domain Routing).

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 else 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 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 accross 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 (*http://aws.amazon.com/ebs/*) 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 only free 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 (*https://aws-portal.amazon.com/gp/aws/html-forms-controller/contactus/ec2-email-limit-rdns-request*).

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 addres. 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.

We can do most of this work we do 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 paritition 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 you don't do that in Ubuntu Lucid 10.04 and later. In previous versions, the boot process continus 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.

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. 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.

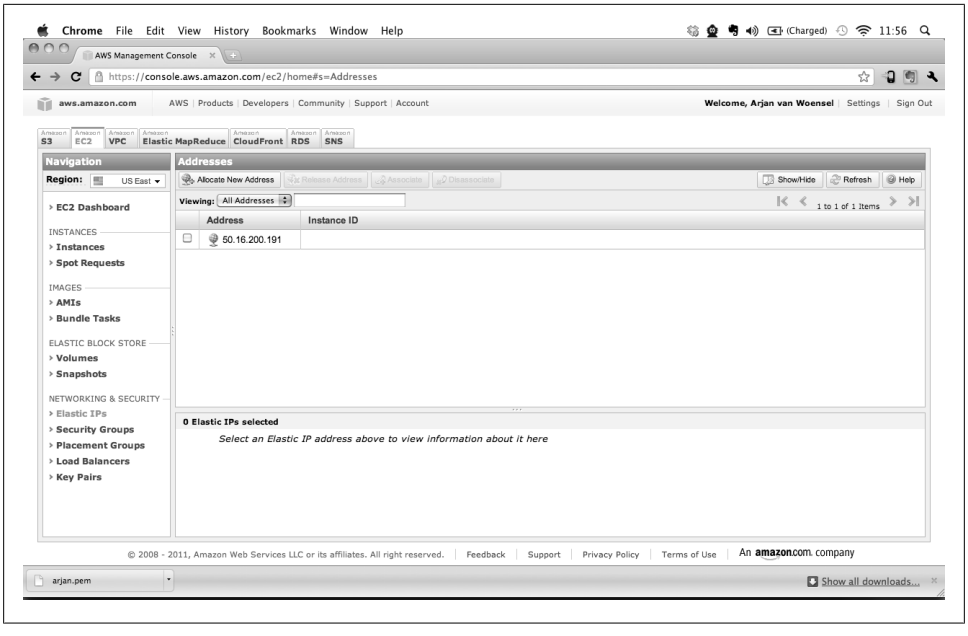

Figure 2-8. AWS Management Console

Install 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
$ 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
# 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 $MAX_TRIES ]; then
            if [ ! -e "$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.

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 can not 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 or if launching a new instance for upgrading is OK for your system, this method is fine.

RDS database

If you have gotten this far, you have probably noticed our love for Amazon RDS. In case you didn't, 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/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, or 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 for you, 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, but here is one 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 readonly 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 (*http://dev.mysql.com/doc/refman/5.1/en/converting-tables-to-innodb.html*).

Before we continue, make sure you sign up for Amazon RDS (*http://aws.amazon.com/rds/*). At this time, almost all of the functionality of RDS is available through the Console. What is missing is important, and we need those features to set up the DB instance the way we want. You already downloaded the Amazon RDS Command Line Toolkit (*http://developer.amazonwebservices.com/connect/entry.jspa?externalID=2928&categoryID=294*) 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.

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 accross 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 somewhere 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 maintence on the other instance.

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

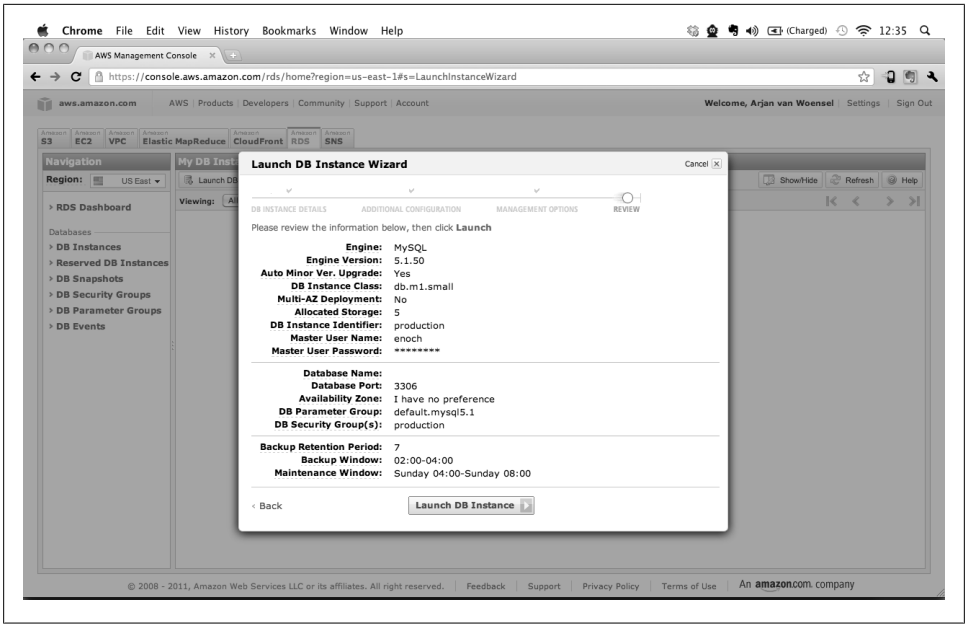

Figure 2-9. Launch the DB instance wizard

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
```

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 Amazon Console looks like Figure 2-10.

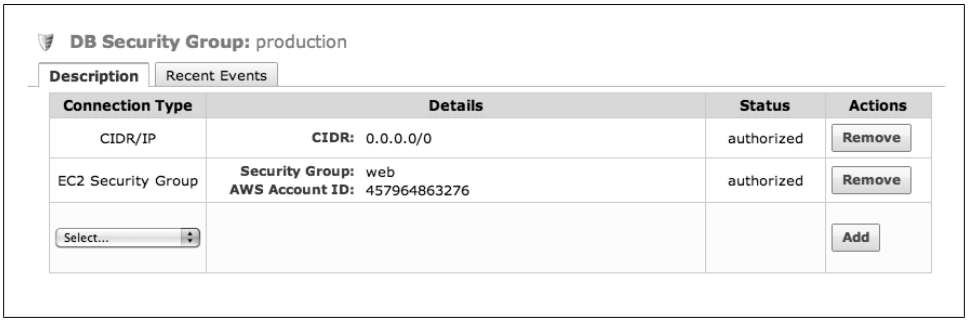

Figure 2-10. Configure a DB security group

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 is this simple, how can it be so good!? There is 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 paramater groups, but it is necessary for some basics like slow query logging.

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:

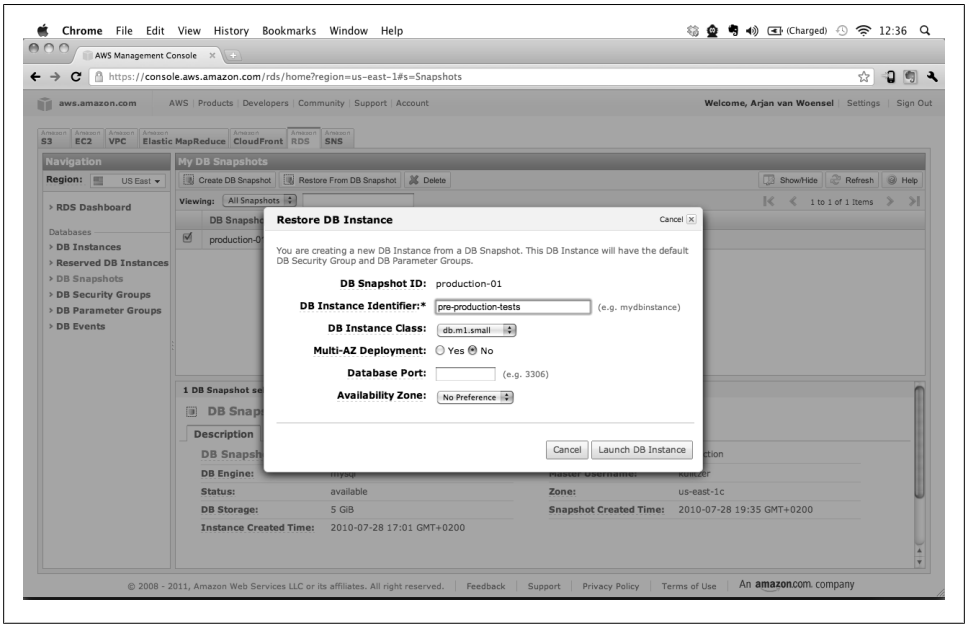

Figure 2-11. Restore DB Instance

```
$ 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.

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-COMMITED 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 level. 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

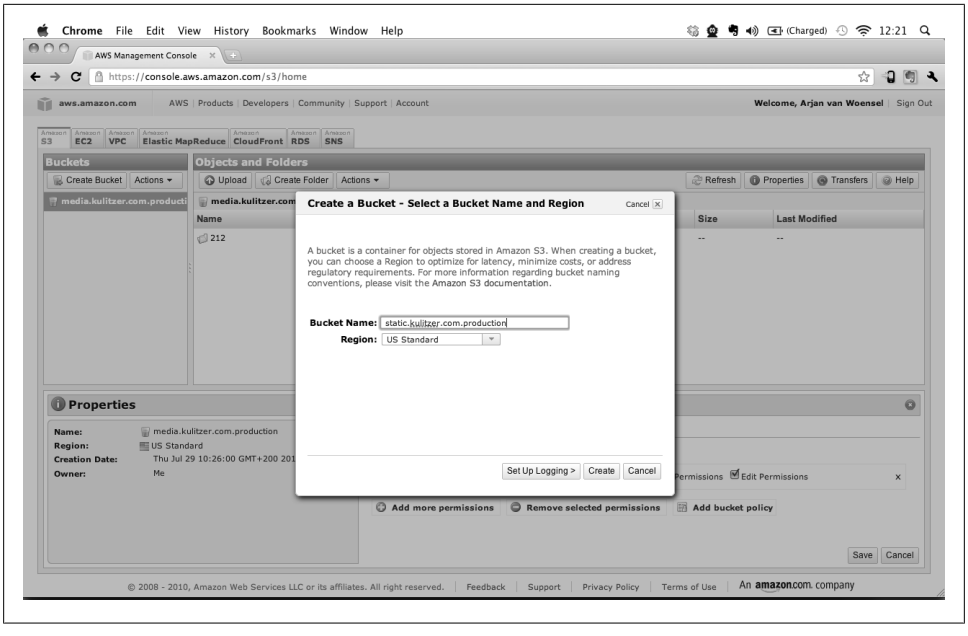

Figure 2-12. Create a bucket

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

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 region), we create our S3 bucket in the same region, though in the Console this region is called US Standard (Figure 2-12).

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 of 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 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.

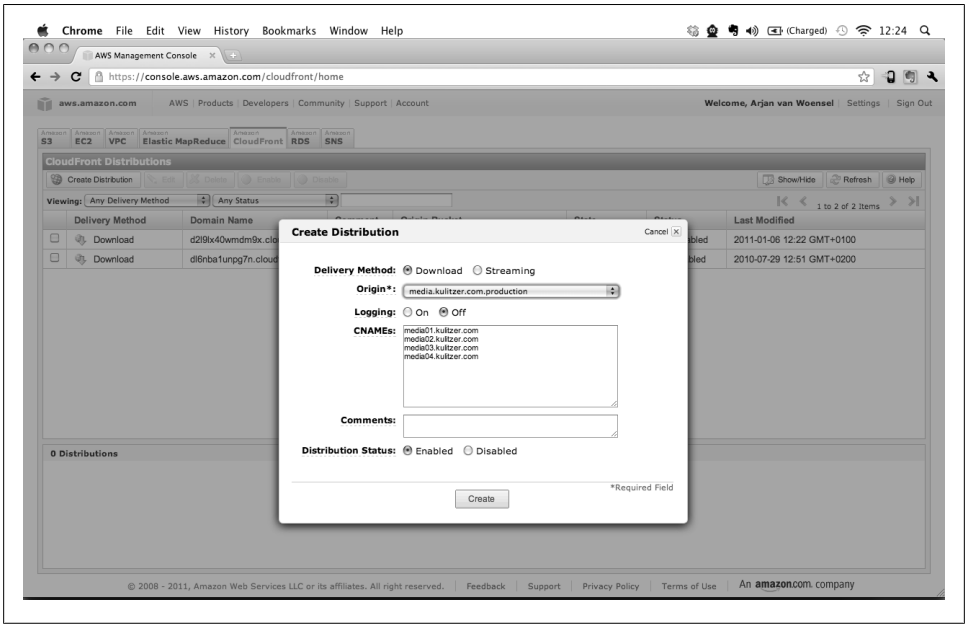

Figure 2-13. Create a distribution

It might take a little while before your distribution is enabled. AWS has already determined the domain for the distribution: `d2l9lx4Owmdm9x.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"
```

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

Figure 2-14. kulitzer.com

from the different domains, allowing your browser to download them in parallel. Figure 2-14 shows our Kulitzer site set up using EC2, RDS, S2, 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 at most within 24 hours, but it is usually much quicker than that. You can override this behaviour 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.

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 audo 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 want to be able to go back in time because you never know. 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 tool to easily, and very quickly, just 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, it is difficult to verify this, because we have never found out where to see how much we use for our snapshots.

Basically, everything else you need to create a sufficient backup/restore mechanism is a way to expire your snapshots. If we can take snapshots for which we can set the expiration date to one week in the future 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.

Installing the Tools

We feel we have to take some time to go over the installation of the SimpleDB client. Not only have we not introduced SimpleDB yet, it uses some tools that are not self explanatory. So, here we go:

1. First, download (but don't install yet) the Perl Library for Amazon SimpleDB.
2. Go to *http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1136*.
3. Look at the prerequites 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, *http://code.google.com/p/amazon-simpledb-cli/*.
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, and that is that it does not accept a region and always takes the default us-east-1 region):

```
$ export AWS_ACCESS_KEY_ID='your acces 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:

```
#!/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.

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, this 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 a three-hour interval, keeping all of them for at least 24 hours, and some up to a year.

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; it 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 group 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 it, 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.

For giving global fast access to static content, we used CloudFront, the content distribution network of AWS. Uploading your 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 your 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 are going to start looking at in the next chapters.