Author Picks

FREE

Cloud Native Applications

Chapters selected by
Michael Wittig and Andreas Wittig

manning

Cloud Native Applications

Selected by Michael Wittig
and Andreas Wittig

Manning Author Picks

To pre-order or learn more about these books go to www.manning.com

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

For more information, please contact

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

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

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

Cover designer: Leslie Haimes

ISBN 9781617294310
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

As engineers, we always aim to improve the feedback loop between us and our customers. This is one reason why we started to use Amazon Web Services four years ago. Being able to automate every step from making changes in our source code to deploying the infrastructure needed to operate our applications was mind blowing. As cloud consultants, we observe that being able to automate and constantly adjust infrastructure is one important reason to use cloud computing for our clients. Another reason why we are focusing on cloud native applications entirely: developing and operating systems that are able to recover from failure automatically was never easier. High Availability is becoming the new normal.

So what has changed? Cloud providers are offering the needed infrastructure for a very reasonable price. AWS, for example, is providing multiple data centers per region that you can use in parallel. Technologies like messaging systems and load balancers allow you to decouple different parts of your system and plan for failure.

We encourage you to read these chapters we've selected from several Manning books and learn how to gain value from cloud computing for your company. This book will guide you through five topics giving you insights into the world of cloud computing. You'll learn how to use Amazon Web Services, one of the most important public cloud providers. You'll get to know to Docker and Mesos as well. Both tools help you to automate and manage your cloud infrastructure. Developing software will generate the biggest value if you adopt your architecture and processes. Decoupling different parts of your system is possible by using a messaging infrastructure, which you'll learn about in the RabbitMQ chapter. In addition, you'll discover different use cases for Netty, a framework that helps you to build solid networking communication into your applications for the cloud.

Amazon Web Services in Action

What is the biggest advantage of using Amazon Web Services (AWS)? For us it's being able to automate every part of your cloud infrastructure. AWS offers an API and lots of tools to launch, configure, modify, and delete computing, storage and networking infrastructure. Our book, *Amazon Web Services in Action*, provides a deep introduction into the most important services and architecture principles. Chapter 1 answers the question: What is Amazon Web Services? You'll learn about the concepts behind AWS and gain a brief overview of what you can do with AWS.

Chapter 1 from *Amazon Web Services in Action* by Michael Wittig and Andreas Wittig.

What is Amazon Web Services?

This chapter covers

- Overview of Amazon Web Services
- Benefits of using Amazon Web Services
- Examples of what you can do with Amazon Web Services
- Creating and setting up an Amazon Web Services account

Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking, at different layers of abstraction. You can use these services to host web sites, run enterprise applications, and mine tremendous amounts of data. The term *web service* means services can be controlled via a web interface. The web interface can be used by machines or by humans via a graphical user interface. The most prominent services are EC2, which offers virtual servers, and S3, which offers storage capacity. Services on AWS work well together; you can use them to replicate your existing on-premises setup or design a new setup from scratch. Services are charged for on a pay-per-use pricing model.

As an AWS customer, you can choose among different data centers. AWS data centers are distributed in the United States, Europe, Asia, and South America. For example, you can start a virtual server in Japan in the same way you can start a virtual server in Ireland. This enables you to serve customers worldwide with a global infrastructure.

The map in figure 1.1 shows the data centers available to all customers.

Which hardware powers AWS?

AWS keeps secret the hardware used in its data centers. The scale at which AWS operates computing, networking, and storage hardware is tremendous. It probably uses commodity components to save money compared to hardware that charges extra for a brand name. Handling of hardware failure is built into real-world processes and software.[1]

AWS also uses hardware especially developed for its use cases. A good example is the Xeon E5-2666 v3 CPU from Intel. This CPU is optimized to power virtual servers from the c4 family.

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

1.1 What is cloud computing?

Almost every IT solution is labeled with the term *cloud computing* or just *cloud* nowadays. A buzzword may help to sell, but it's hard to work with in a book.

Cloud computing, or the cloud, is a metaphor for supply and consumption of IT resources. The IT resources in the cloud aren't directly visible to the user; there are layers of abstraction in between. The level of abstraction offered by the cloud may vary

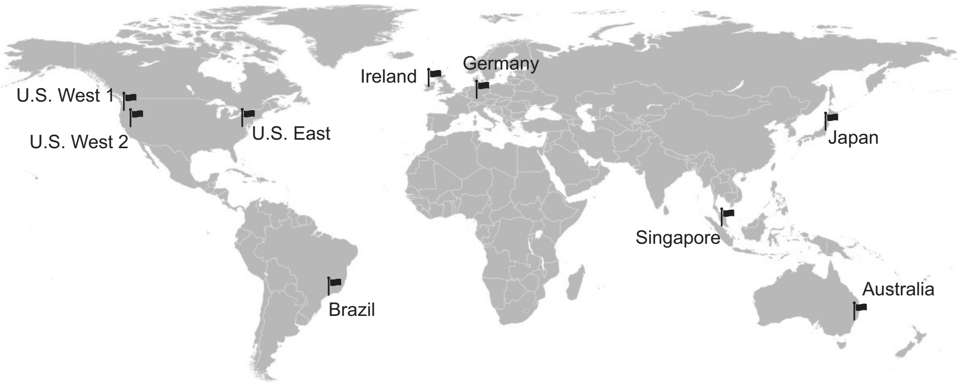


Figure 1.1 AWS data center locations

[1] Bernard Golden, "Amazon Web Services (AWS) Hardware," *For Dummies*, http://mng.bz/k6lT.

from virtual hardware to complex distributed systems. Resources are available on demand in enormous quantities and paid for per use.

Here's a more official definition from the National Institute of Standards and Technology:

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

Clouds are often divided into the following types:

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

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

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

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

1.2 What can you do with AWS?

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

1.2.1 Hosting a web shop

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

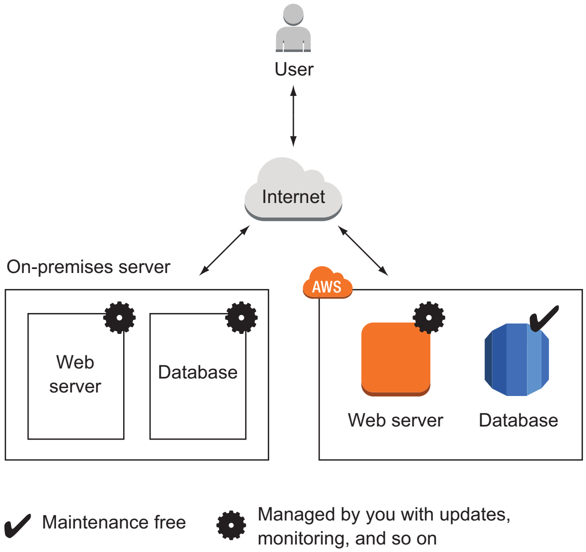


Figure 1.2 Running a web shop on-premises vs. on AWS

John realized that other options are available to improve his setup on AWS with additional services:

- The web shop consists of dynamic content (such as products and their prices) and static content (such as the company logo). By splitting dynamic and static content, John reduced the load for his web servers and improved performance by delivering the static content over a content delivery network (CDN).
- John uses maintenance-free services including a database, an object store, and a DNS system on AWS. This frees him from managing these parts of the system, decreases operational costs, and improves quality.
- The application running the web shop can be installed on virtual servers. John split the capacity of the old on-premises server into multiple smaller virtual servers at no extra cost. If one of these virtual servers fails, the load balancer will send customer requests to the other virtual servers. This setup improves the web shop's reliability.

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

John started a proof-of-concept project and found that his web application can be transferred to AWS and that services are available to help improve his setup.

1.2.2 Running a Java EE application in your private network

Maureen is a senior system architect in a global corporation. She wants to move parts of the business applications to AWS when the company's data-center contract expires in a few months, to reduce costs and gain flexibility. She found that it's possible to run enterprise applications on AWS.

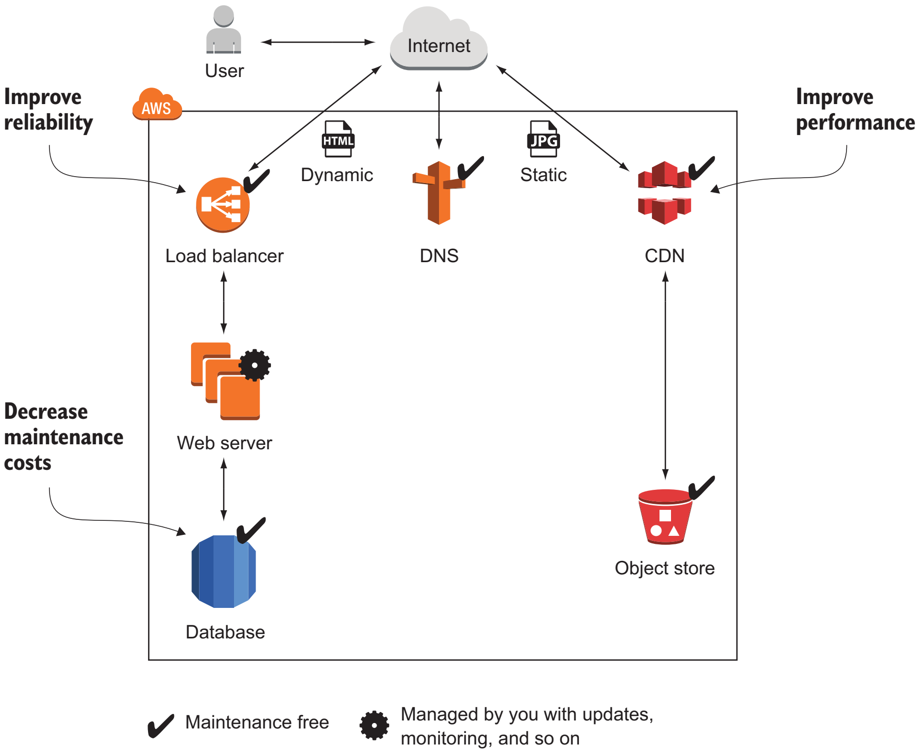


Figure 1.3 Running a web shop on AWS with CDN for better performance, a load balancer for high availability, and a managed database to decrease maintenance costs

To do so, she defines a virtual network in the cloud and connects it to the corporate network through a virtual private network (VPN) connection. The company can control access and protect mission-critical data by using subnets and control traffic between them with access-control lists. Maureen controls traffic to the internet using Network Address Translation (NAT) and firewalls. She installs application servers on virtual machines (VMs) to run the Java EE application. Maureen is also thinking about storing data in a SQL database service (such as Oracle Database Enterprise Edition or Microsoft SQL Server EE). Figure 1.4 illustrates Maureen's architecture.

Maureen has managed to connect the on-premises data center with a private network on AWS. Her team has already started to move the first enterprise application to the cloud.

1.2.3 Meeting legal and business data archival requirements

Greg is responsible for the IT infrastructure of a small law office. His primary goal is to store and archive all data in a reliable and durable way. He operates a file server to

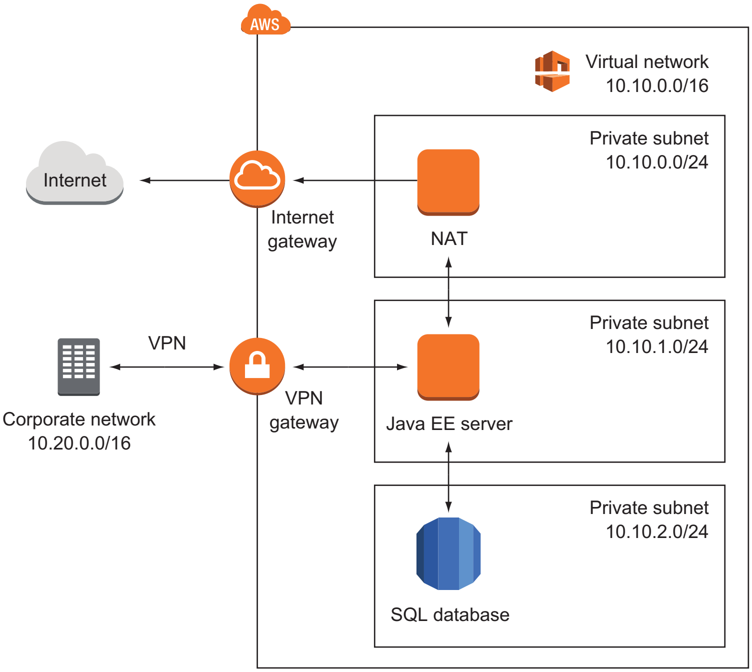


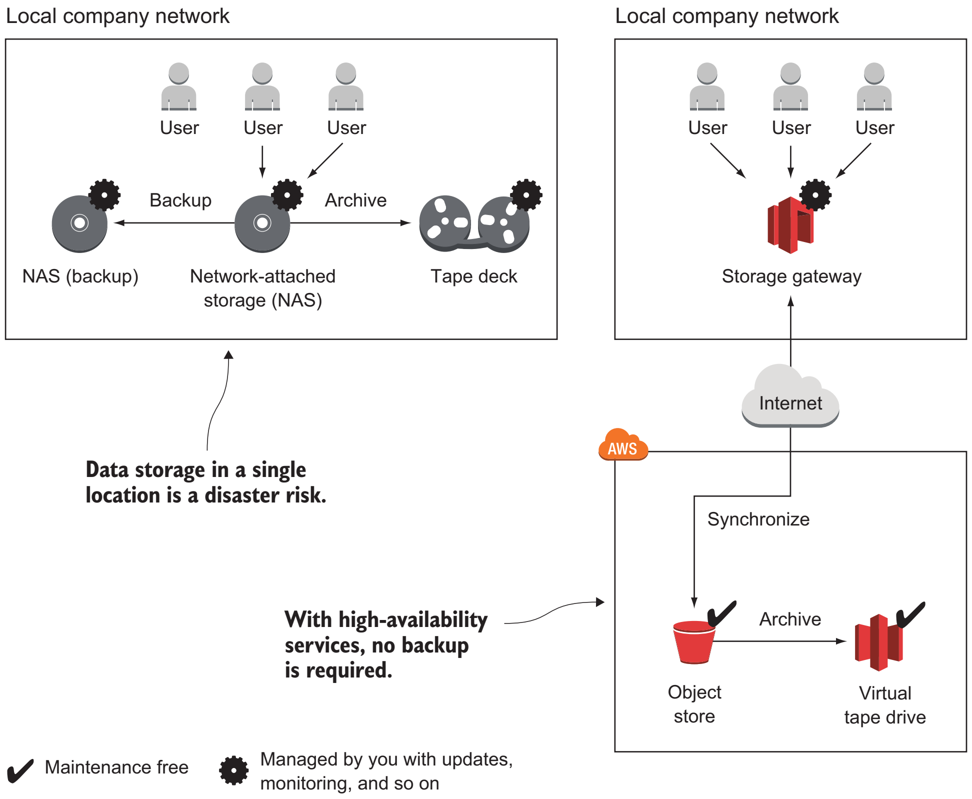
Figure 1.4 Running a Java EE application with enterprise networking on AWS

offer the possibility of sharing documents within the office. Storing all the data is a challenge for him:

- He needs to back up all files to prevent the loss of critical data. To do so, Greg copies the data from the file server to another network-attached storage, so he had to buy the hardware for the file server twice. The file server and the backup server are located close together, so he is failing to meet disaster-recovery requirements to recover from a fire or a break-in.
- To meet legal and business data archival requirements, Greg needs to store data for a long time. Storing data for 10 years or longer is tricky. Greg uses an expensive archive solution to do so.

To save money and increase data security, Greg decided to use AWS. He transferred data to a highly available object store. A storage gateway makes it unnecessary to buy and operate network-attached storage and a backup on-premises. A virtual tape deck takes over the task of archiving data for the required length of time. Figure 1.5 shows how Greg implemented this use case on AWS and compares it to the on-premises solution.

Greg is fine with the new solution to store and archive data on AWS because he was able to improve quality and he gained the possibility of scaling storage size.



Data storage in a single location is a disaster risk.

With high-availability services, no backup is required.

Figure 1.5 Backing up and archiving data on-premises and on AWS

1.2.4 *Implementing a fault-tolerant system architecture*

Alexa is a software engineer working for a fast-growing startup. She knows that Murphy's Law applies to IT infrastructure: anything that can go wrong, will go wrong. Alexa is working hard to build a fault-tolerant system to prevent outages from ruining the business. She knows that there are two type of services on AWS: fault-tolerant services and services that can be used in a fault-tolerant way. Alexa builds a system like the one shown in figure 1.6 with a fault-tolerant architecture. The database service is offered with replication and failover handling. Alexa uses virtual servers acting as web servers. These virtual servers aren't fault tolerant by default. But Alexa uses a load balancer and can launch multiple servers in different data centers to achieve fault tolerance.

So far, Alexa has protected the startup from major outages. Nevertheless, she and her team are always planning for failure.

You now have a broad idea of what you can do with AWS. Generally speaking, you can host any application on AWS. The next section explains the nine most important benefits AWS has to offer.

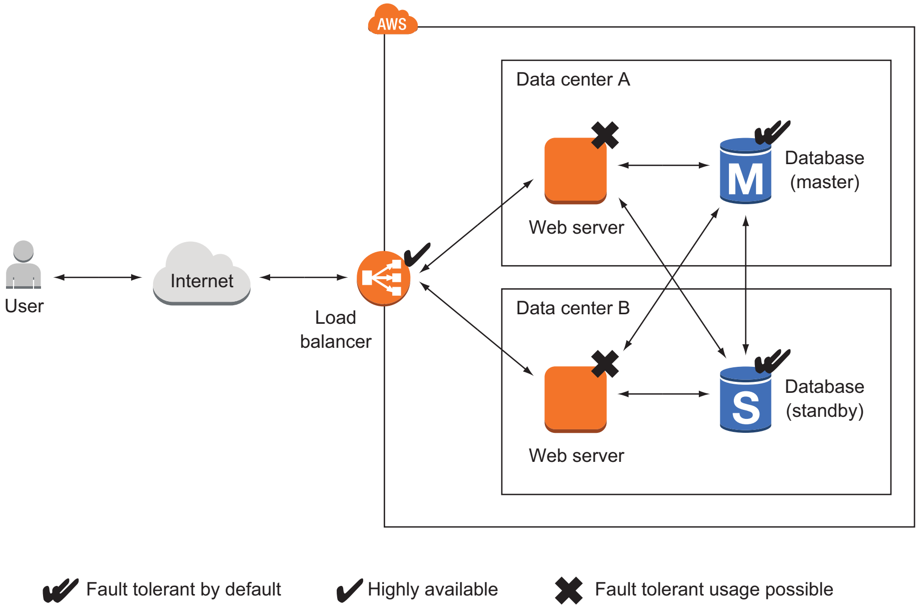


Figure 1.6 Building a fault-tolerant system on AWS

1.3 How you can benefit from using AWS

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

1.3.1 Innovative and fast-growing platform

In 2014, AWS announced more than 500 new services and features during its yearly conference, re:Invent at Las Vegas. On top of that, new features and improvements are released every week. You can transform these new services and features into innovative solutions for your customers and thus achieve a competitive advantage.

The number of attendees to the re:Invent conference grew from 9,000 in 2013 to 13,500 in 2014.[1] AWS counts more than 1 million businesses and government agencies among its customers, and in its Q1 2014 results discussion, the company said it will continue to hire more talent to grow even further.[2] You can expect even more new features and services in the coming years.

[1] Greg Bensinger, "Amazon Conference Showcases Another Side of the Retailer's Business," *Digits*, Nov. 12, 2014, http://mng.bz/hTBo.

[2] "Amazon.com's Management Discusses Q1 2014 Results - Earnings Call Transcript," *Seeking Alpha*, April 24, 2014, http://mng.bz/60qX.

1.3.2 Services solve common problems

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

1.3.3 Enabling automation

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

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

1.3.4 Flexible capacity (scalability)

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

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

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

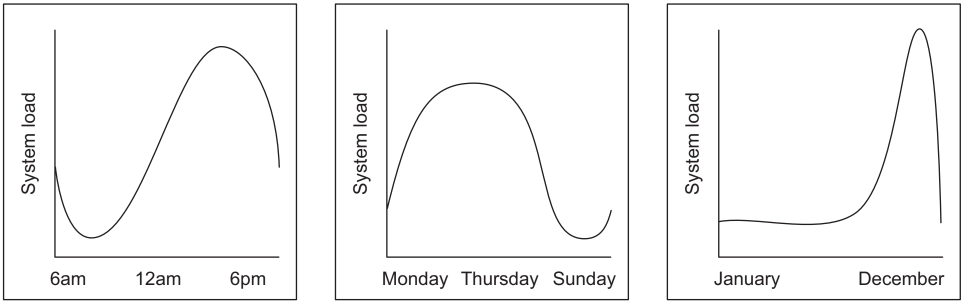


Figure 1.7 Seasonal traffic patterns for a web shop

Flexible capacity also means you can shut down unused systems. In one of our last projects, the test environment only ran from 7:00 a.m. to 8:00 p.m. on weekdays, allowing us to save 60%.

1.3.5 Built for failure (reliability)

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

1.3.6 Reducing time to market

In AWS, you request a new virtual server, and a few minutes later that virtual server is booted and ready to use. The same is true with any other AWS service available. You can use them all on demand. This allows you to adapt your infrastructure to new requirements very quickly.

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

1.3.7 Benefiting from economies of scale

At the time of writing, the charges for using AWS have been reduced 42 times since 2008:

- In December 2014, charges for outbound data transfer were lowered by up to 43%.
- In November 2014, charges for using the search service were lowered by 50%.
- In March 2014, charges for using a virtual server were lowered by up to 40%.

As of December 2014, AWS operated 1.4 million servers. All processes related to operations must be optimized to operate at that scale. The bigger AWS gets, the lower the prices will be.

1.3.8 Worldwide

You can deploy your applications as close to your customers as possible. AWS has data centers in the following locations:

- United States (northern Virginia, northern California, Oregon)
- Europe (Germany, Ireland)
- Asia (Japan, Singapore)
- Australia
- South America (Brazil)

With AWS, you can run your business all over the world.

1.3.9 Professional partner

AWS is compliant with the following:

- *ISO 27001*—A worldwide information security standard certified by an independent and accredited certification body

- *FedRAMP & DoD CSM*—Ensures secure cloud computing for the U.S. Federal Government and the U.S. Department of Defense
- *PCI DSS Level 1*—A data security standard (DSS) for the payment card industry (*PCI*) to protect cardholders data
- *ISO 9001*—A standardized quality management approach used worldwide and certified by an independent and accredited certification body

If you're still not convinced that AWS is a professional partner, you should know that Airbnb, Amazon, Intuit, NASA, Nasdaq, Netflix, SoundCloud, and many more are running serious workloads on AWS.

The cost benefit is elaborated in more detail in the next section.

1.4 How much does it cost?

A bill from AWS is similar to an electric bill. Services are billed based on usage. You pay for the hours a virtual server was running, the used storage from the object store (in gigabytes), or the number of running load balancers. Services are invoiced on a monthly basis. The pricing for each service is publicly available; if you want to calculate the monthly cost of a planned setup, you can use the AWS Simple Monthly Calculator (http://aws.amazon.com/calculator).

1.4.1 Free Tier

You can use some AWS services for free during the first 12 months after you sign up. The idea behind the Free Tier is to enable you to experiment with AWS and get some experience. Here is what's included in the Free Tier:

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

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

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

1.4.2 Billing example

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

- *Based on hours of usage*—If you use a server for 61 minutes, that's usually counted as 2 hours.

- *Based on traffic*—Traffic can be measured in gigabytes or in number of requests.
- *Based on storage usage*—Usage can be either provisioned capacity (for example, 50 GB volume no matter how much you use) or real usage (such as 2.3 GB used).

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

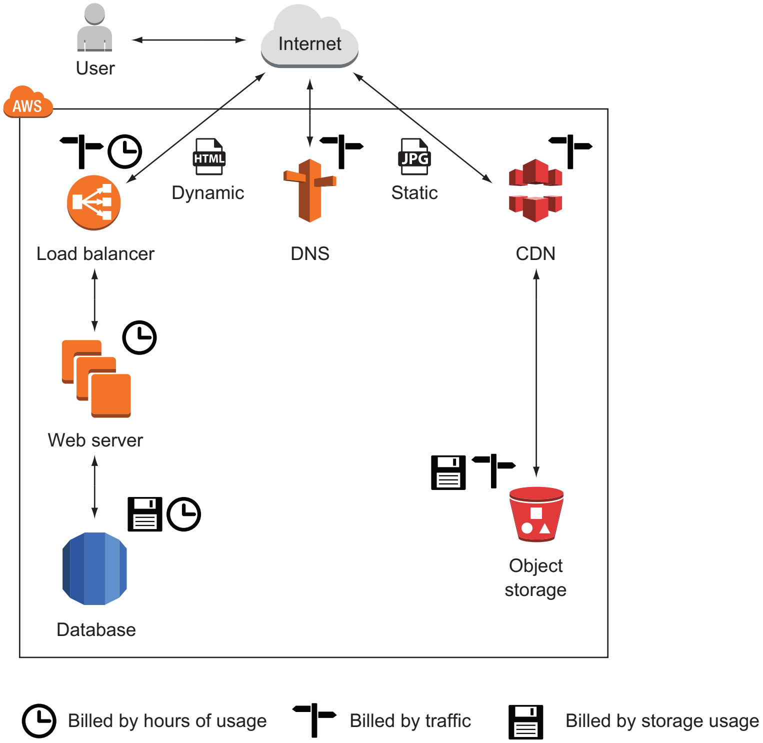


Figure 1.8 Web shop billing example

Let's assume your web shop started successfully in January, and you decided to run a marketing campaign to increase sales for the next month. Lucky you: you were able to increase the number of visitors of your web shop fivefold in February. As you already know, you have to pay for AWS based on usage. Table 1.1 shows your bills for January and February. The number of visitors increased from 100,000 to 500,000, and your monthly bill increased from 142.37 USD to 538.09 USD, which is a 3.7-fold increase. Because your web shop had to handle more traffic, you had to pay more for services, such as the CDN, the web servers, and the database. Other services, like the storage of static files, didn't experience more usage, so the price stayed the same.

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

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

Service	January usage	February usage	February charge	Increase
Visits to website	100,000	500,000		
CDN	26 M requests + 25 GB traffic	131 M requests + 125 GB traffic	113.31 USD	90.64 USD
Static files	50 GB used storage	50 GB used storage	1.50 USD	0.00 USD
Load balancer	748 hours + 50 GB traffic	748 hours + 250 GB traffic	20.30 USD	1.60 USD
Web servers	1 server = 748 hours	4 servers = 2,992 hours	204.96 USD	153.72 USD
Database (748 hours)	Small server + 20 GB storage	Large server + 20 GB storage	170.66 USD	128.10 USD
Traffic (outgoing traffic to internet)	51 GB	255 GB	22.86 USD	18.46 USD
DNS	2 M requests	10 M requests	4.50 USD	3.20 USD
Total cost			538.09 USD	395.72 USD

1.4.3 Pay-per-use opportunities

The AWS pay-per-use pricing model creates new opportunities. You no longer need to make upfront investments in infrastructure. You can start servers on demand and only pay per hour of usage; and you can stop using those servers whenever you like and no longer have to pay for them. You don't need to make an upfront commitment regarding how much storage you'll use.

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

1.5 Comparing alternatives

AWS isn't the only cloud computing provider. Microsoft and Google have cloud offerings as well.

OpenStack is different because it's open source and developed by more than 200 companies including IBM, HP, and Rackspace. Each of these companies uses OpenStack to operate its own cloud offerings, sometimes with closed source add-ons. You could run your own cloud based on OpenStack, but you would lose most of the benefits outlined in section 1.3.

Comparing cloud providers isn't easy, because open standards are mostly missing. Functionality like virtual networks and message queuing are realized differently. If you know what features you need, you can compare the details and make your decision.

Otherwise, AWS is your best bet because the chances are highest that you'll find a solution for your problem.

Following are some common features of cloud providers:

- Virtual servers (Linux and Windows)
- Object store
- Load balancer
- Message queuing
- Graphical user interface
- Command-line interface

The more interesting question is, how do cloud providers differ? Table 1.2 compares AWS, Azure, Google Cloud Platform, and OpenStack.

Table 1.2 Differences between AWS, Microsoft Azure, Google Cloud Platform, and OpenStack

	AWS	Azure	Google Cloud Platform	OpenStack
Number of services	Most	Many	Enough	Few
Number of locations (multiple data centers per location)	9	13	3	Yes (depends on the OpenStack provider)
Compliance	Common standards (ISO 27001, HIPAA, FedRAMP, SOC), IT Grundschutz (Germany), G-Cloud (UK)	Common standards (ISO 27001, HIPAA, FedRAMP, SOC), ISO 27018 (cloud privacy), G-Cloud (UK)	Common standards (ISO 27001, HIPAA, FedRAMP, SOC)	Yes (depends on the OpenStack provider)
SDK languages	Android, Browsers (JavaScript), iOS, Java, .NET, Node.js (JavaScript), PHP, Python, Ruby, Go	Android, iOS, Java, .NET, Node.js (JavaScript), PHP, Python, Ruby	Java, Browsers (JavaScript), .NET, PHP, Python	-
Integration into development process	Medium, not linked to specific ecosystems	High, linked to the Microsoft ecosystem (for example, .NET development)	High, linked to the Google ecosystem (for example, Android)	-
Block-level storage (attached via network)	Yes	Yes (can be used by multiple virtual servers simultaneously)	No	Yes (can be used by multiple virtual servers simultaneously)
Relational database	Yes (MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server)	Yes (Azure SQL Database, Microsoft SQL Server)	Yes (MySQL)	Yes (depends on the OpenStack provider)
NoSQL database	Yes (proprietary)	Yes (proprietary)	Yes (proprietary)	No
DNS	Yes	No	Yes	No

Table 1.2 Differences between AWS, Microsoft Azure, Google Cloud Platform, and OpenStack *(continued)*

	AWS	Azure	Google Cloud Platform	OpenStack
Virtual network	Yes	Yes	No	Yes
Pub/sub messaging	Yes (proprietary, JMS library available)	Yes (proprietary)	Yes (proprietary)	No
Machine-learning tools	Yes	Yes	Yes	No
Deployment tools	Yes	Yes	Yes	No
On-premises datacenter integration	Yes	Yes	Yes	No

In our opinion, AWS is the most mature cloud platform available at the moment.

1.6 Exploring AWS services

Hardware for computing, storing, and networking is the foundation of the AWS cloud. AWS runs software services on top of the hardware to provide the cloud, as shown in figure 1.9. A web interface, the API, acts as an interface between AWS services and your applications.

You can manage services by sending requests to the API manually via a GUI or programmatically via a SDK. To do so, you can use a tool like the Management Console, a web-based user interface, or a command-line tool. Virtual servers have a peculiarity: you can connect to virtual servers through SSH, for example, and gain administrator

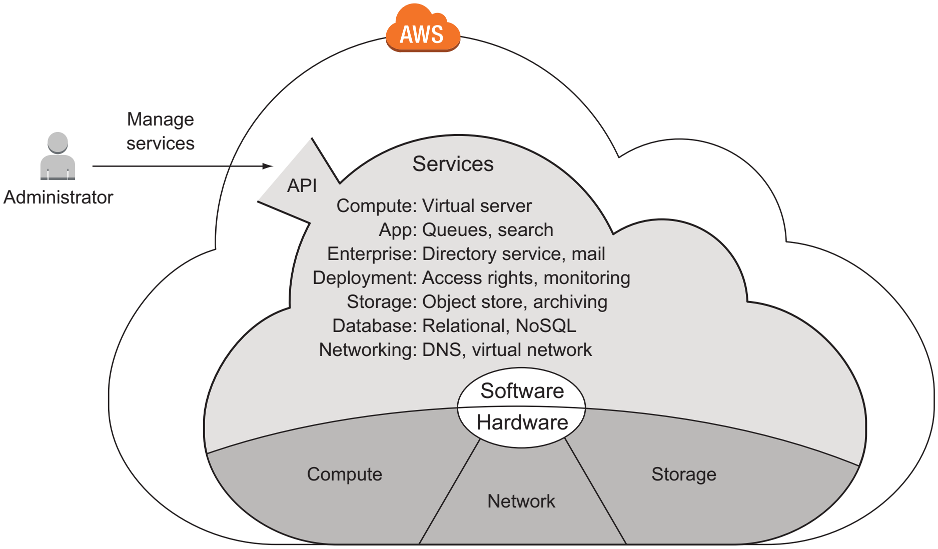

AWS
Manage services
Administrator
API
Services
Compute: Virtual server
App: Queues, search
Enterprise: Directory service, mail
Deployment: Access rights, monitoring
Storage: Object store, archiving
Database: Relational, NoSQL
Networking: DNS, virtual network
Software
Hardware
Compute
Network
Storage

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

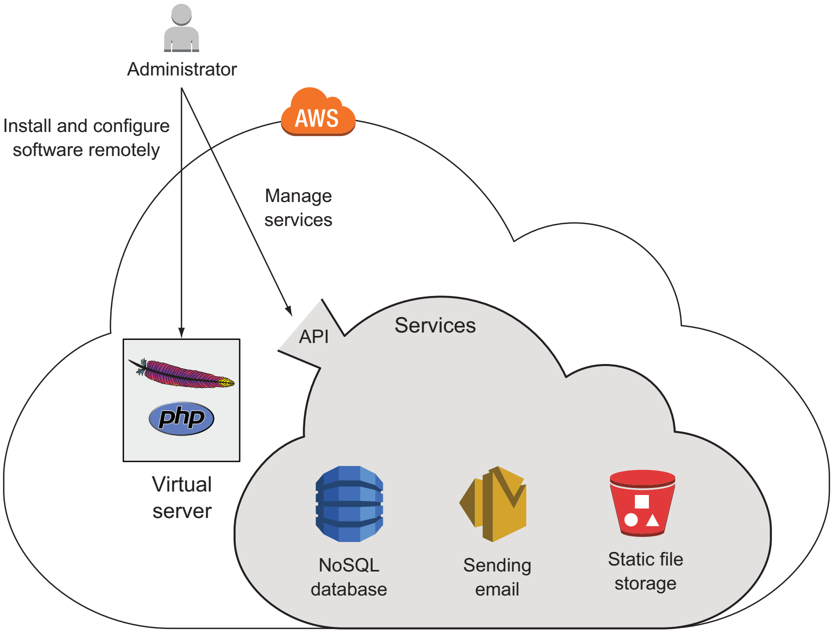


Figure 1.10 Managing a custom application running on a virtual server and dependent services

access. This means you can install any software you like on a virtual server. Other services, like the NoSQL database service, offer their features through an API and hide everything that's going on behind the scenes. Figure 1.10 shows an administrator installing a custom PHP web application on a virtual server and managing dependent services such as a NoSQL database used by the PHP web application.

Users send HTTP requests to a virtual server. A web server is installed on this virtual server along with a custom PHP web application. The web application needs to talk to AWS services in order to answer HTTP requests from users. For example, the web application needs to query data from a NoSQL database, store static files, and send email. Communication between the web application and AWS services is handled by the API, as figure 1.11 shows.

The number of different services available can be scary at the outset. The following categorization of AWS services will help you to find your way through the jungle:

- *Compute services* offer computing power and memory. You can start virtual servers and use them to run your applications.
- *App services* offer solutions for common use cases like message queues, topics, and searching large amounts of data to integrate into your applications.

- *Enterprise services* offer independent solutions such as mail servers and directory services.
- *Deployment and administration services* work on top of the services mentioned so far. They help you grant and revoke access to cloud resources, monitor your virtual servers, and deploy applications.
- *Storage* is needed to collect, persist, and archive data. AWS offers different storage options: an object store or a network-attached storage solution for use with virtual servers.
- *Database storage* has some advantages over simple storage solutions when you need to manage structured data. AWS offers solutions for relational and NoSQL databases.
- *Networking services* are an elementary part of AWS. You can define private networks and use a well-integrated DNS.

Be aware that we cover only the most important categories and services here. Other services are available, and you can also run your own applications on AWS.

Now that we've looked at AWS services in detail, it's time for you to learn how to interact with those services.

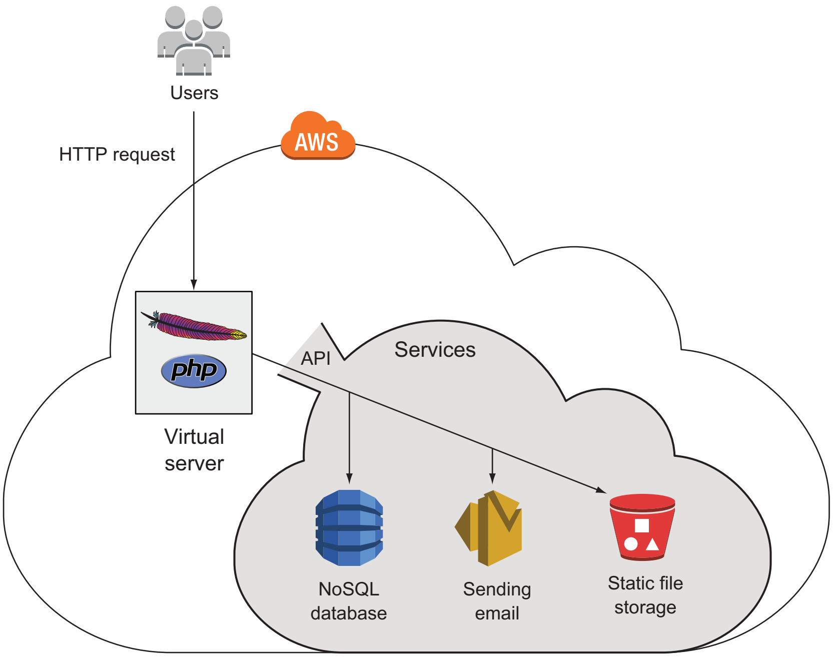


Figure 1.11 Handling an HTTP request with a custom web application using additional AWS services

1.7 Interacting with AWS

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

Next, we'll give you an overview of the tools available to make calls to the AWS API. You can compare the ability of these tools to automate your daily tasks.

1.7.1 Management Console

You can use the web-based Management Console to interact with AWS. You can manually control AWS with this convenient GUI, which runs in every modern web browser (Chrome, Firefox, Safari ≥ 5, IE ≥ 9); see figure 1.13.

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

1.7.2 Command-line interface

You can start a virtual server, create storage, and send email from the command line. With the command-line interface (CLI), you can control everything on AWS; see figure 1.14.

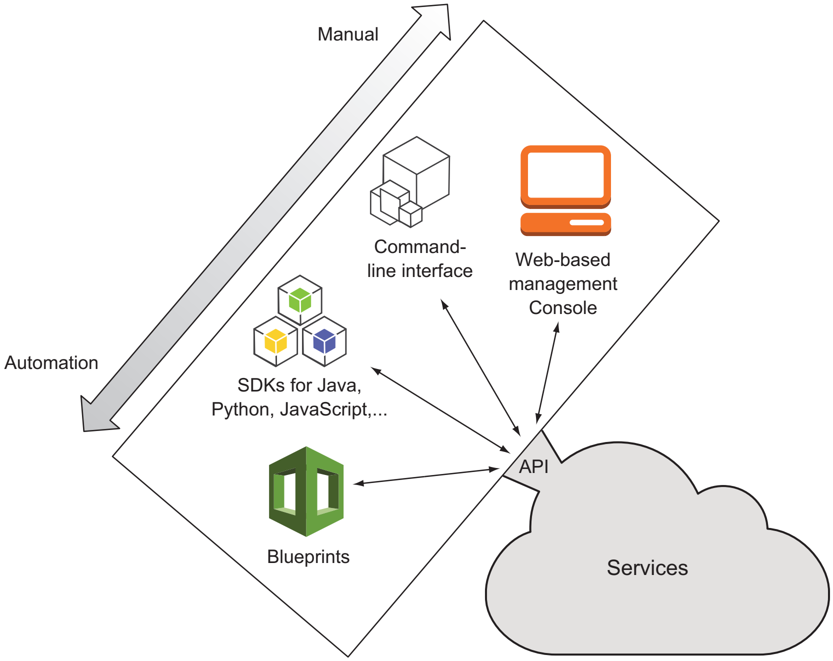
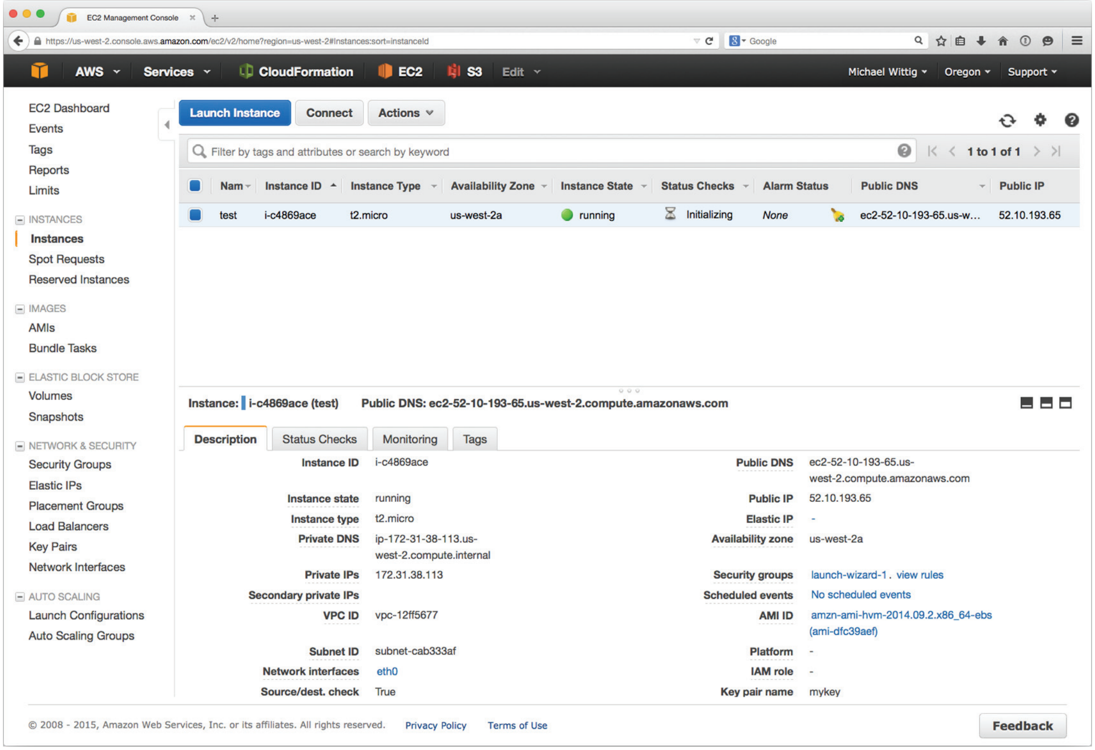

Manual
Automation
Command-line interface
Web-based management Console
SDKs for Java, Python, JavaScript,...
Blueprints
API
Services

Figure 1.12 Tools to interact with the AWS API



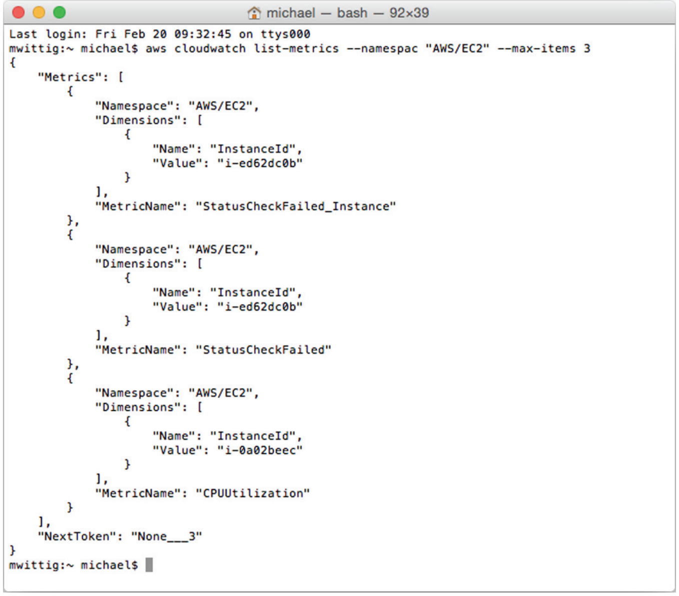
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS	Public IP
test	i-c4869ace	t2.micro	us-west-2a	running	Initializing	None	ec2-52-10-193-65.us-w...	52.10.193.65

Instance: i-c4869ace (test) Public DNS: ec2-52-10-193-65.us-west-2.compute.amazonaws.com

Instance ID	i-c4869ace	Public DNS	ec2-52-10-193-65.us-west-2.compute.amazonaws.com
Instance state	running	Public IP	52.10.193.65
Instance type	t2.micro	Elastic IP	-
Private DNS	ip-172-31-38-113.us-west-2.compute.internal	Availability zone	us-west-2a
Private IPs	172.31.38.113	Security groups	launch-wizard-1 . view rules
Secondary private IPs		Scheduled events	No scheduled events
VPC ID	vpc-12ff5677	AMI ID	amzn-ami-hvm-2014.09.2.x86_64-ebs (ami-dfc39aef)
Subnet ID	subnet-cab333af	Platform	-
Network interfaces	eth0	IAM role	-
Source/dest. check	True	Key pair name	mykey

© 2008 - 2015, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Figure 1.13 Management Console



```
Last login: Fri Feb 20 09:32:45 on ttys000
mwittig:~ michael$ aws cloudwatch list-metrics --namespac "AWS/EC2" --max-items 3
{
    "Metrics": [
        {
            "Namespace": "AWS/EC2",
            "Dimensions": [
                {
                    "Name": "InstanceId",
                    "Value": "i-ed62dc0b"
                }
            ],
            "MetricName": "StatusCheckFailed_Instance"
        },
        {
            "Namespace": "AWS/EC2",
            "Dimensions": [
                {
                    "Name": "InstanceId",
                    "Value": "i-ed62dc0b"
                }
            ],
            "MetricName": "StatusCheckFailed"
        },
        {
            "Namespace": "AWS/EC2",
            "Dimensions": [
                {
                    "Name": "InstanceId",
                    "Value": "i-0a02beec"
                }
            ],
            "MetricName": "CPUUtilization"
        }
    ],
    "NextToken": "None___3"
}
mwittig:~ michael$ 
```

Figure 1.14 Command-line interface

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

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

1.7.3 SDKs

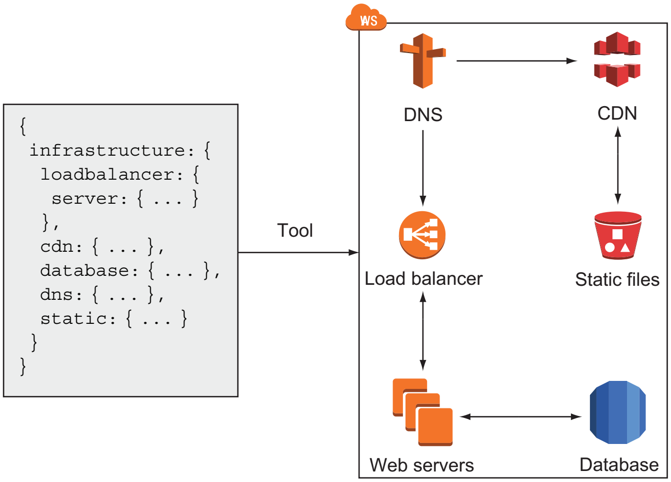
Sometimes you need to call AWS from within your application. With SDKs, you can use your favorite programming language to integrate AWS into your application logic. AWS provides SDKs for the following:

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

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

1.7.4 Blueprints

A *blueprint* is a description of your system containing all services and dependencies. The blueprint doesn't say anything about the necessary steps or the order to achieve the described system. Figure 1.15 shows how a blueprint is transferred into a running system.



```
{
 infrastructure: {
  loadbalancer: {
   server: { ... }
  },
  cdn: { ... },
  database: { ... },
  dns: { ... },
  static: { ... }
 }
}
```

Figure 1.15 Infrastructure automation with blueprints

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

A blueprint removes much of the burden from you because you no longer need to worry about dependencies during system creation—the blueprint automates the entire process. You'll learn more about automating your infrastructure in chapter 4.

It's time to get started creating your AWS account and exploring AWS practice after all that theory.

1.8 Creating an AWS account

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

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

Using an old account?

You can use your existing AWS account while working on the examples in this book. In this case, your usage may not be covered by the Free Tier, and you may have to pay for your usage.

Also, if you created your existing AWS account before December 4, 2013, you should create a new one: there are legacy issues that may cause trouble when you try our examples.

1.8.1 Signing up

The sign-up process consists of five steps:

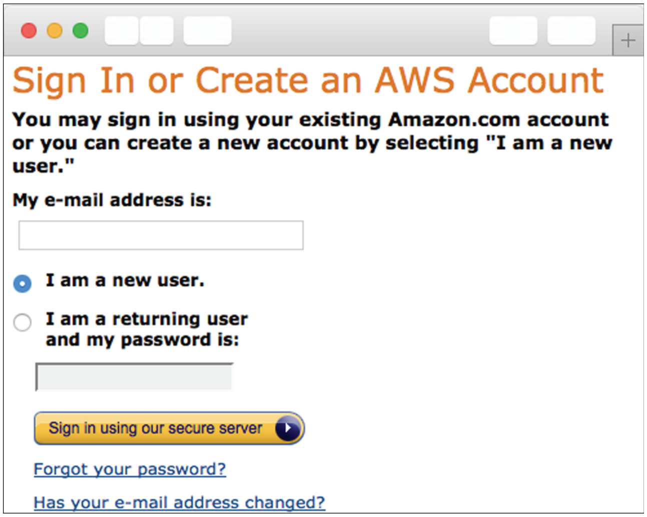
1. Provide your login credentials.
2. Provide your contact information.
3. Provide your payment details.
4. Verify your identity.
5. Choose your support plan.

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

1. PROVIDING YOUR LOGIN CREDENTIALS

The Sign Up page, shown in figure 1.16, gives you two choices. You can either create an account using your Amazon.com account or create an account from scratch. If you create the account from scratch, follow along. Otherwise, skip to step 5.

Fill in your email address, and select I Am a New User. Go on to the next step to create your login credentials. We advise you to choose a strong password to prevent misuse



Sign In or Create an AWS Account

You may sign in using your existing Amazon.com account or you can create a new account by selecting "I am a new user."

My e-mail address is:

● I am a new user.

○ I am a returning user and my password is:

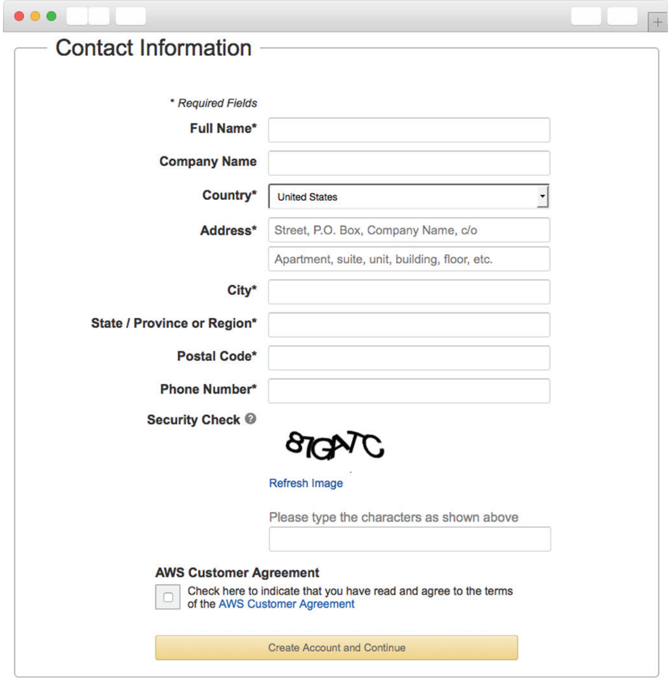
Sign in using our secure server
Forgot your password?
Has your e-mail address changed?

Figure 1.16 Creating an AWS account: Sign Up page

of your account. We suggest a password with 16 characters, numbers, and symbols. If someone gets access to your account, they can destroy your systems or steal your data.

2. Providing your contact information

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



Contact Information

* Required Fields

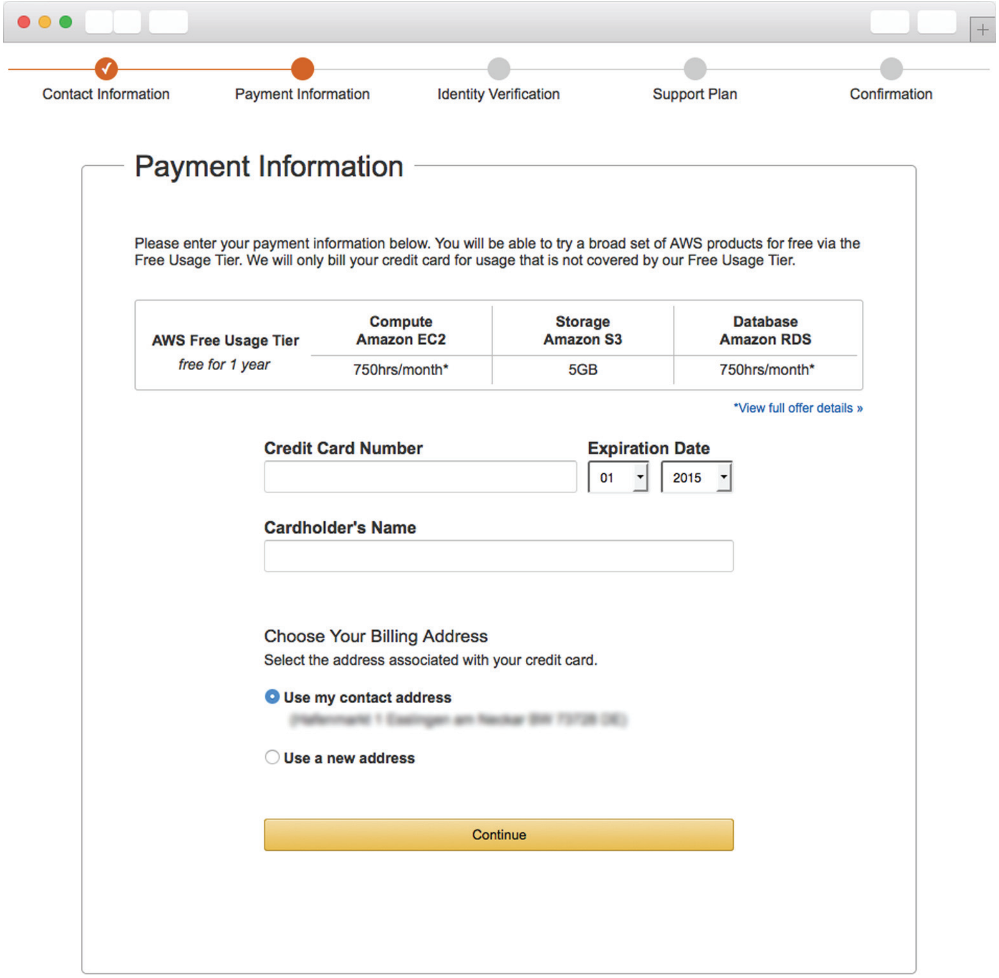
Full Name*	
Company Name	
Country*	United States
Address*	Street, P.O. Box, Company Name, c/o Apartment, suite, unit, building, floor, etc.
City*	
State / Province or Region*	
Postal Code*	
Phone Number*	
Security Check	87GNTC

Please type the characters as shown above

AWS Customer Agreement

□ Check here to indicate that you have read and agree to the terms of the AWS Customer Agreement

Figure 1.17 Creating an AWS account: providing your contact information



Payment Information

Please enter your payment information below. You will be able to try a broad set of AWS products for free via the Free Usage Tier. We will only bill your credit card for usage that is not covered by our Free Usage Tier.

AWS Free Usage Tier *free for 1 year*	Compute Amazon EC2	Storage Amazon S3	Database Amazon RDS
	750hrs/month*	5GB	750hrs/month*

*View full offer details »

Credit Card Number

Expiration Date 01 2015

Cardholder's Name

Choose Your Billing Address

Select the address associated with your credit card.

● Use my contact address

[redacted]

○ Use a new address

Continue

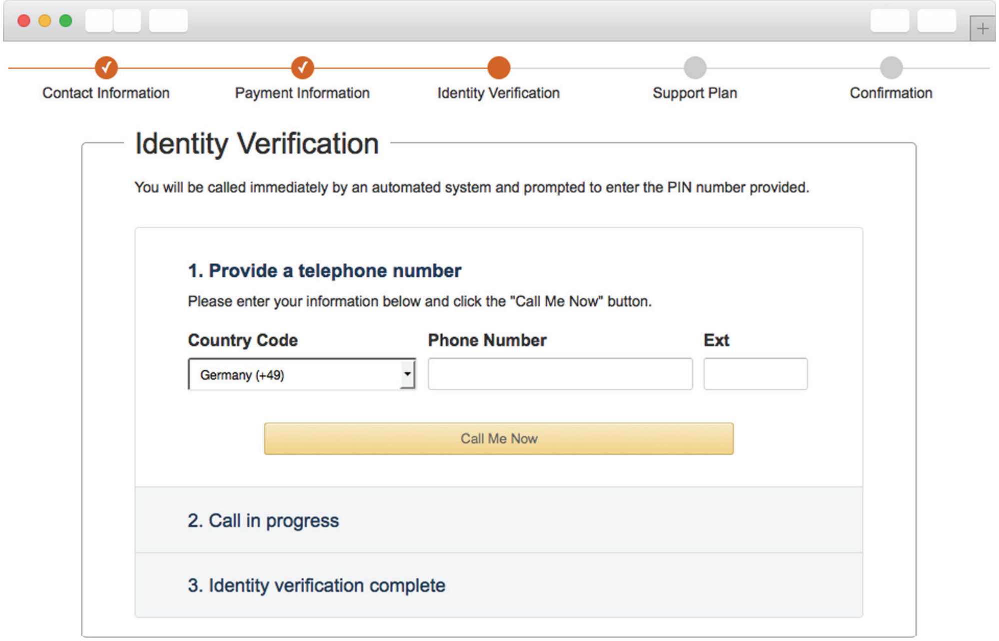
Figure 1.18 Creating an AWS account: providing your payment details

3. Provide your payment details

Now the screen shown in figure 1.18 asks for your payment information. AWS supports MasterCard and Visa. You can set your preferred payment currency later, if you don't want to pay your bills in USD; supported currencies are EUR, GBP, CHF, AUD, and some others.

4. Verifying your identity

The next step is to verify your identity. Figure 1.19 shows the first step of the process.



Identity Verification

You will be called immediately by an automated system and prompted to enter the PIN number provided.

1. Provide a telephone number

Please enter your information below and click the "Call Me Now" button.

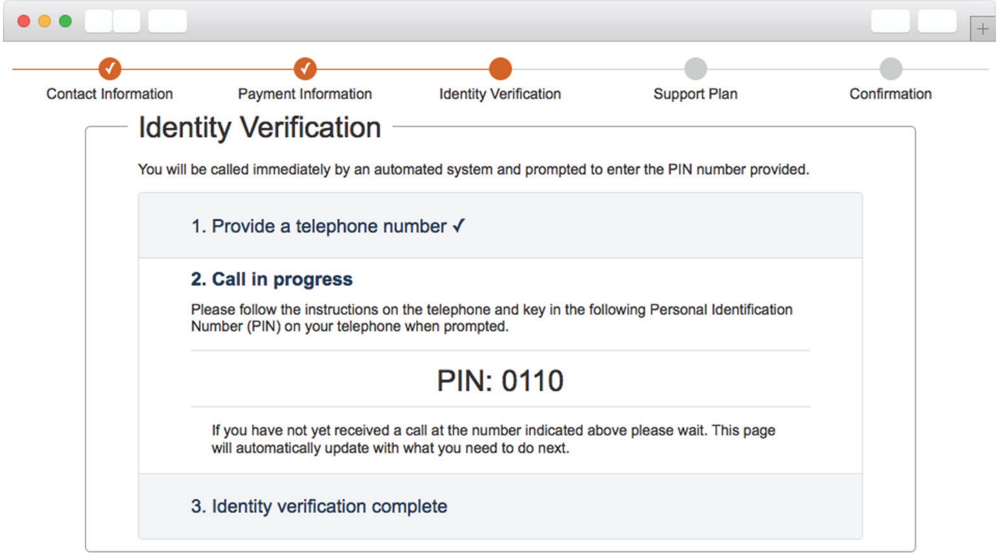
Country Code	Phone Number	Ext
Germany (+49)		

2. Call in progress

3. Identity verification complete

Figure 1.19 Creating an AWS account: verifying your identity (1 of 2)

After you complete the first part, you'll receive a call from AWS. A robot voice will ask you for your PIN, which will be like the one shown in figure 1.20. Your identity will be verified, and you can continue with the last step.



Identity Verification

You will be called immediately by an automated system and prompted to enter the PIN number provided.

1. Provide a telephone number ✓

2. Call in progress

Please follow the instructions on the telephone and key in the following Personal Identification Number (PIN) on your telephone when prompted.

PIN: 0110

If you have not yet received a call at the number indicated above please wait. This page will automatically update with what you need to do next.

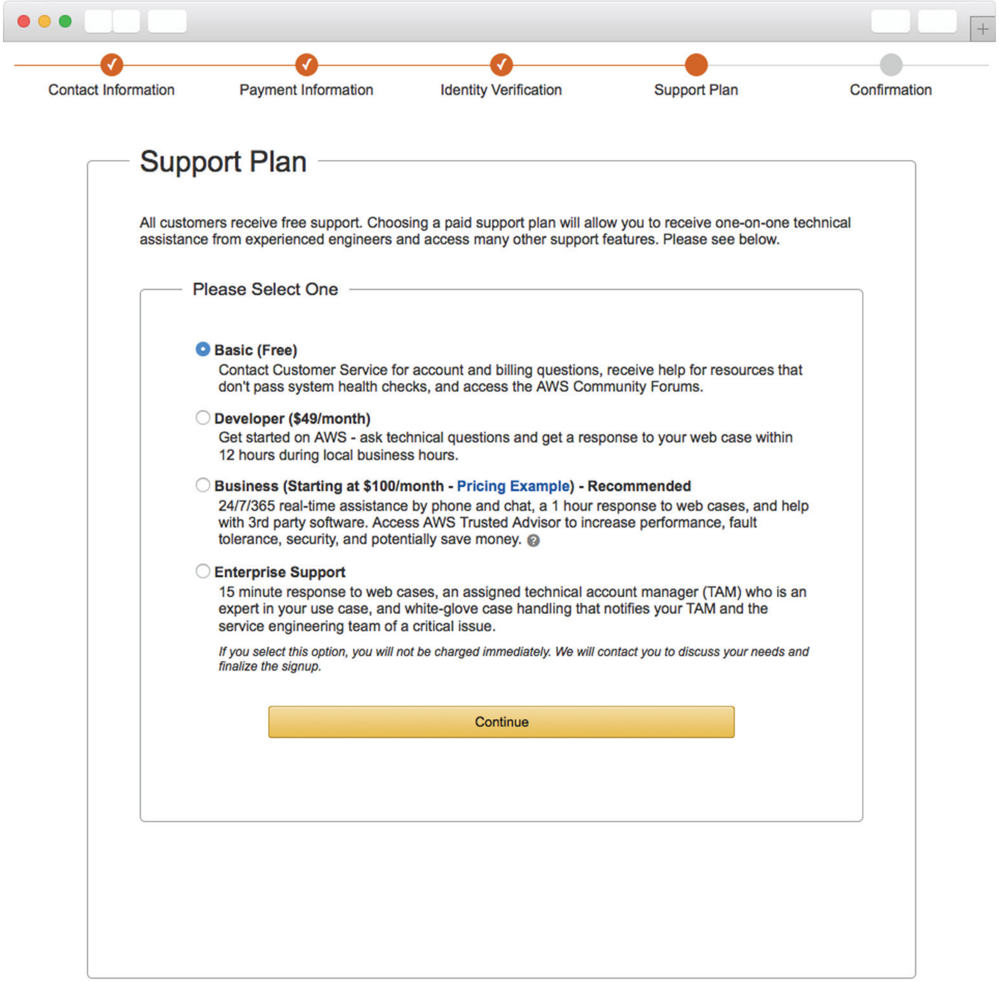
3. Identity verification complete

Figure 1.20 Creating an AWS account: verifying your identity (2 of 2)

5. Choosing your support plan

The last step is to choose a support plan; see figure 1.21. In this case, select the Basic plan, which is free. If you later create an AWS account for your business, we recommend the Business support plan. You can even switch support plans later.

High five! You're done. Now you can log in to your account with the AWS Management Console.



Support Plan

All customers receive free support. Choosing a paid support plan will allow you to receive one-on-one technical assistance from experienced engineers and access many other support features. Please see below.

Please Select One

● **Basic (Free)**
Contact Customer Service for account and billing questions, receive help for resources that don't pass system health checks, and access the AWS Community Forums.

○ **Developer ($49/month)**
Get started on AWS - ask technical questions and get a response to your web case within 12 hours during local business hours.

○ **Business (Starting at $100/month - Pricing Example) - Recommended**
24/7/365 real-time assistance by phone and chat, a 1 hour response to web cases, and help with 3rd party software. Access AWS Trusted Advisor to increase performance, fault tolerance, security, and potentially save money.

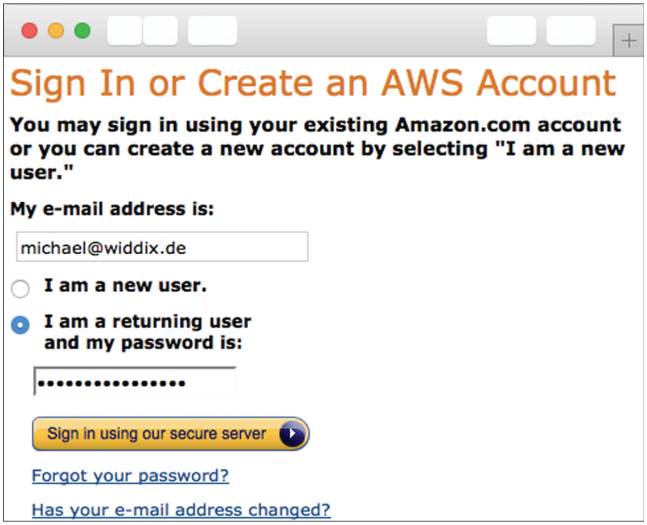
○ **Enterprise Support**
15 minute response to web cases, an assigned technical account manager (TAM) who is an expert in your use case, and white-glove case handling that notifies your TAM and the service engineering team of a critical issue.

If you select this option, you will not be charged immediately. We will contact you to discuss your needs and finalize the signup.

Figure 1.21 Creating an AWS account: choosing your support plan

1.8.2 Signing In

You have an AWS account and are ready to sign in to the AWS Management Console at https://console.aws.amazon.com. As mentioned earlier, the Management Console is a web-based tool you can use to control AWS resources. The Management Console



Sign In or Create an AWS Account

You may sign in using your existing Amazon.com account or you can create a new account by selecting "I am a new user."

My e-mail address is: michael@widdix.de

○ **I am a new user.**

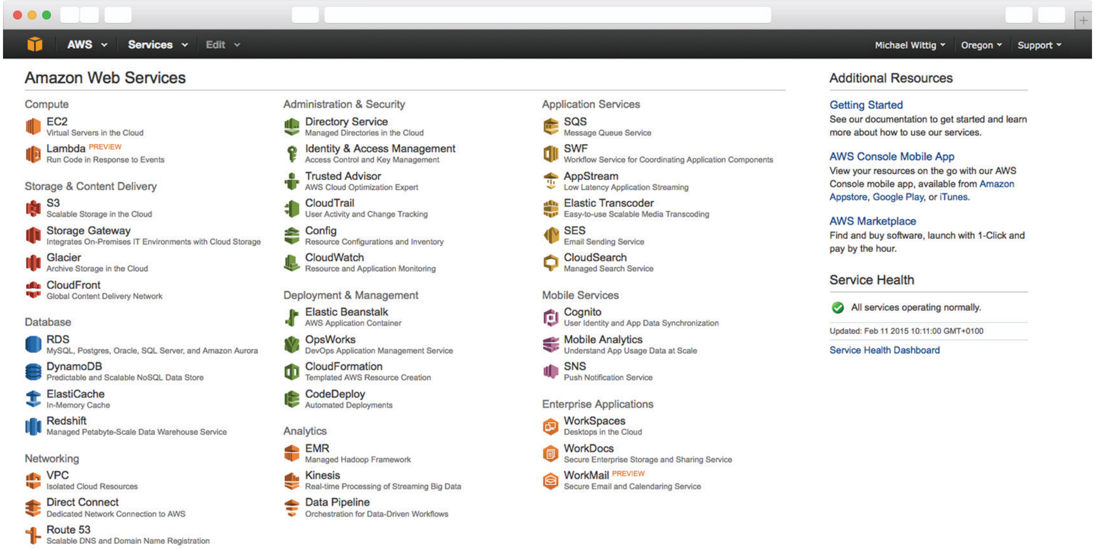
● **I am a returning user and my password is:** ••••••••••••••••

Sign in using our secure server
Forgot your password?
Has your e-mail address changed?

Figure 1.22 Sign in to the Management Console.

uses the AWS API to make most of the functionality available to you. Figure 1.22 shows the Sign In page.

Enter your login credentials and click Sign In Using Our Secure Server to see the Management Console, shown in figure 1.23.



Amazon Web Services

Compute
- EC2 — Virtual Servers in the Cloud
- Lambda PREVIEW — Run Code in Response to Events

Storage & Content Delivery
- S3 — Scalable Storage in the Cloud
- Storage Gateway — Integrates On-Premises IT Environments with Cloud Storage
- Glacier — Archive Storage in the Cloud
- CloudFront — Global Content Delivery Network

Database
- RDS — MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- DynamoDB — Predictable and Scalable NoSQL Data Store
- ElastiCache — In-Memory Cache
- Redshift — Managed Petabyte-Scale Data Warehouse Service

Networking
- VPC — Isolated Cloud Resources
- Direct Connect — Dedicated Network Connection to AWS
- Route 53 — Scalable DNS and Domain Name Registration

Administration & Security
- Directory Service — Managed Directories in the Cloud
- Identity & Access Management — Access Control and Key Management
- Trusted Advisor — AWS Cloud Optimization Expert
- CloudTrail — User Activity and Change Tracking
- Config — Resource Configurations and Inventory
- CloudWatch — Resource and Application Monitoring

Deployment & Management
- Elastic Beanstalk — AWS Application Container
- OpsWorks — DevOps Application Management Service
- CloudFormation — Templated AWS Resource Creation
- CodeDeploy — Automated Deployments

Analytics
- EMR — Managed Hadoop Framework
- Kinesis — Real-time Processing of Streaming Big Data
- Data Pipeline — Orchestration for Data-Driven Workflows

Application Services
- SQS — Message Queue Service
- SWF — Workflow Service for Coordinating Application Components
- AppStream — Low Latency Application Streaming
- Elastic Transcoder — Easy-to-use Scalable Media Transcoding
- SES — Email Sending Service
- CloudSearch — Managed Search Service

Mobile Services
- Cognito — User Identity and App Data Synchronization
- Mobile Analytics — Understand App Usage Data at Scale
- SNS — Push Notification Service

Enterprise Applications
- WorkSpaces — Desktops in the Cloud
- WorkDocs — Secure Enterprise Storage and Sharing Service
- WorkMail PREVIEW — Secure Email and Calendaring Service

Additional Resources

Getting Started
See our documentation to get started and learn more about how to use our services.

AWS Console Mobile App
View your resources on the go with our AWS Console mobile app, available from Amazon Appstore, Google Play, or iTunes.

AWS Marketplace
Find and buy software, launch with 1-Click and pay by the hour.

Service Health

✓ All services operating normally.

Updated: Feb 11 2015 10:11:00 GMT+0100

Service Health Dashboard

Figure 1.23 AWS Management Console

The most important part is the navigation bar at the top; see figure 1.24. It consists of six sections:

- *AWS*—Gives you a fast overview of all resources in your account.
- *Services*—Provides access to all AWS services.
- *Custom section (Edit)*—Click Edit and drag-and-drop important services here to personalize the navigation bar.
- *Your name*—Lets you access billing information and your account, and also lets you sign out.
- *Your region*—Lets you choose your region. You'll learn about regions in section 3.5. You don't need to change anything here now.
- *Support*—Gives you access to forums, documentation, and a ticket system.

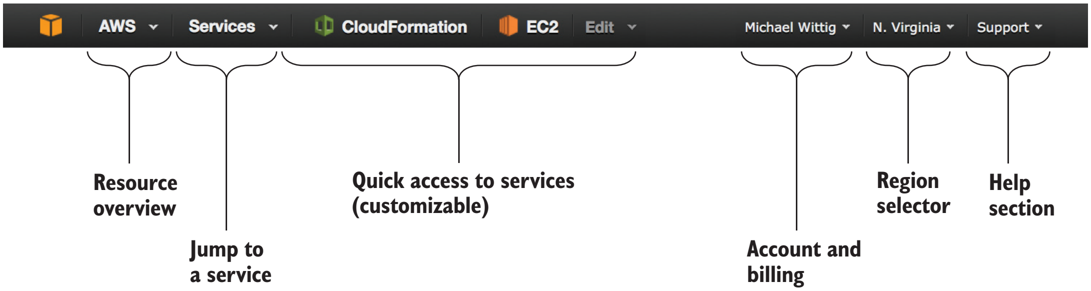


Figure 1.24 AWS Management Console navigation bar

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

1.8.3 Creating a key pair

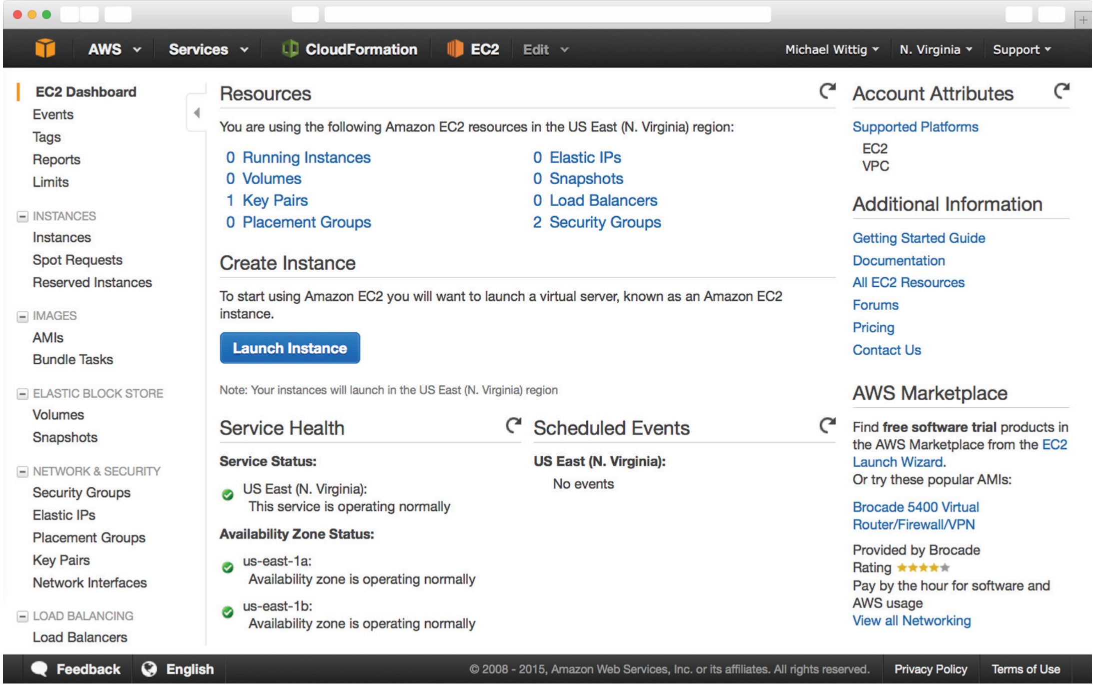
To access a virtual server in AWS, you need a *key pair* consisting of a private key and a public key. The public key will be uploaded to AWS and inserted into the virtual server. The private key is yours; it's like your password, but much more secure. Protect your private key as if it's a password. It's your secret, so don't lose it—you can't retrieve it.

To access a Linux server, you use the SSH protocol; you'll authenticate with the help of your key pair instead of a password during login. You access a Windows server via Remote Desktop Protocol (RDP); you'll need your key pair to decrypt the administrator password before you can log in.

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

1. Open the AWS Management Console at https://console.aws.amazon.com.
2. Click Services in the navigation bar, find the EC2 service, and click it.
3. Your browser shows the EC2 Management Console.

The EC2 Management Console, shown in figure 1.25, is split into three columns. The first column is the EC2 navigation bar; because EC2 is one of the oldest services, it has many



Resources

You are using the following Amazon EC2 resources in the US East (N. Virginia) region:

0 Running Instances
0 Volumes
1 Key Pairs
0 Placement Groups
0 Elastic IPs
0 Snapshots
0 Load Balancers
2 Security Groups

Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Note: Your instances will launch in the US East (N. Virginia) region

Service Health

Service Status:

US East (N. Virginia):
This service is operating normally

Availability Zone Status:

us-east-1a:
Availability zone is operating normally

us-east-1b:
Availability zone is operating normally

Scheduled Events

US East (N. Virginia):
No events

Account Attributes

Supported Platforms
EC2
VPC

Additional Information

Getting Started Guide
Documentation
All EC2 Resources
Forums
Pricing
Contact Us

AWS Marketplace

Find **free software trial** products in the AWS Marketplace from the EC2 Launch Wizard.
Or try these popular AMIs:

Brocade 5400 Virtual Router/Firewall/VPN

Provided by Brocade
Rating ★★★★★
Pay by the hour for software and AWS usage
View all Networking

© 2008 - 2015, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 1.25 EC2 Management Console

features that you can access via the navigation bar. The second column gives you a brief overview of all your EC2 resources. The third column provides additional information.

Follow these steps to create a new key pair:

1. Click Key Pairs in the navigation bar under Network & Security.
2. Click the Create Key Pair button on the page shown in figure 1.26.
3. Name the Key Pair `mykey`. If you choose another name, you must replace the name in all the following examples!

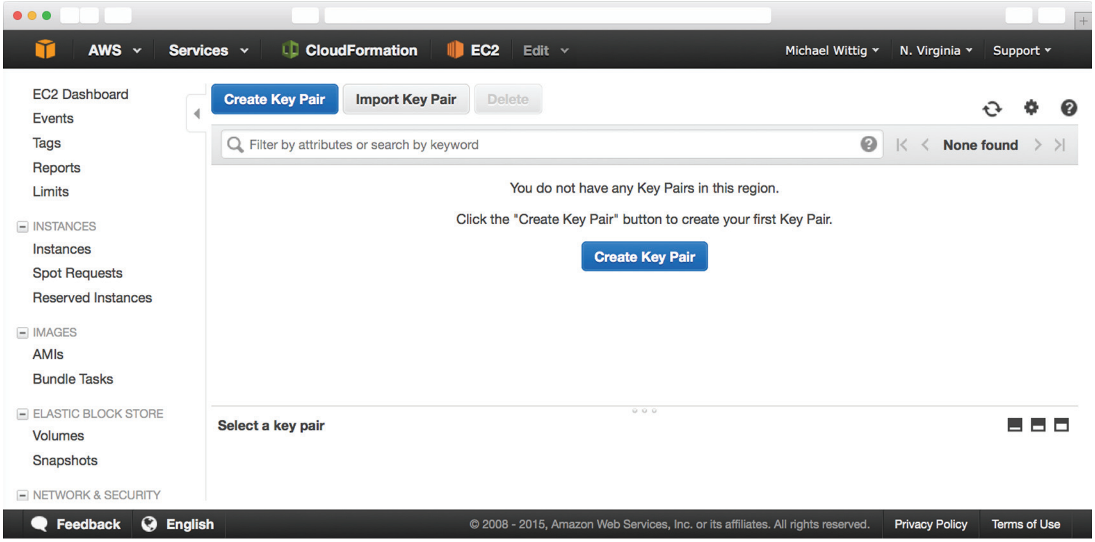
During key-pair creation, you downloaded a file called mykey.pem. You must now prepare that key for future use. Depending on your operating system, you may need to do things differently, so please read the section that fits your OS.

Using your own key pair

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

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

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



You do not have any Key Pairs in this region.

Click the "Create Key Pair" button to create your first Key Pair.

Figure 1.26 EC2 Management Console key pairs

LINUX AND MAC OS X

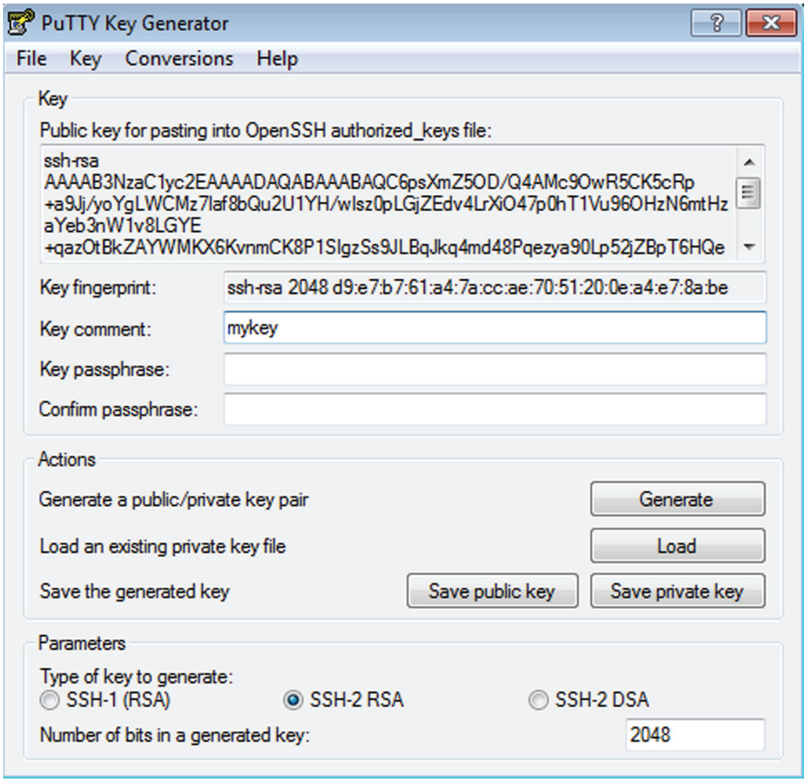
The only thing you need to do is change the access rights of mykey.pem so that only you can read the file. To do so, run `chmod 400 mykey.pem` in the terminal. You'll learn about how to use your key when you need to log in to a virtual server for the first time in this book.

WINDOWS

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

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

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



Public key for pasting into OpenSSH authorized_keys file:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC6psXmZ5OD/Q4AMc9OwR5CK5cRp
+a9Jj/yoYgLWCMz7laf8bQu2U1YH/wlsz0pLGjZEdv4LrXiO47p0hT1Vu96OHzN6mtHz
aYeb3nW1v8LGYE
+qazOtBkZAYWMKX6KvnmCK8P1SlgzSs9JLBqJkq4md48Pqezya90Lp52jZBpT6HQe
```

Key fingerprint: ssh-rsa 2048 d9:e7:b7:61:a4:7a:cc:ae:70:51:20:0e:a4:e7:8a:be

Key comment: mykey

Key passphrase:

Confirm passphrase:

Generate a public/private key pair

Load an existing private key file

Save the generated key

Type of key to generate:
○ SSH-1 (RSA) ● SSH-2 RSA ○ SSH-2 DSA

Number of bits in a generated key: 2048

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

1.8.4 Creating a billing alarm

Before you use your AWS account in the next chapter, we advise you to create a billing alarm. If you exceed the Free Tier, an email is sent to you. The book warns you whenever an example isn't covered by the Free Tier. Please make sure that you carefully follow the cleanup steps after each example. To make sure you haven't missed something during cleanup, please create a billing alarm as advised by AWS: http://mng.bz/M7Sj.

1.9 Summary

- Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking that work well together.
- Cost savings aren't the only benefit of using AWS. You'll also profit from an innovative and fast-growing platform with flexible capacity, fault-tolerant services, and a worldwide infrastructure.
- Any use case can be implemented on AWS, whether it's a widely used web application or a specialized enterprise application with an advanced networking setup.

- You can interact with AWS in many different ways. You can control the different services by using the web-based GUI; use code to manage AWS programmatically from the command line or SDKs; or use blueprints to set up, modify, or delete your infrastructure on AWS.
- Pay-per-use is the pricing model for AWS services. Computing power, storage, and networking services are billed similarly to electricity.
- Creating an AWS account is easy. Now you know how to set up a key pair so you can log in to virtual servers for later use.

Physical data centers require lots of equipment and take time and resources to manage. If you need a data center, but don’t want to build your own, Amazon Web Services may be your solution. Whether you’re analyzing real-time data, building software as a service, or running an e-commerce site, AWS offers you a reliable cloud-based platform with services that scale.

Amazon Web Services in Action introduces you to computing, storing, and networking in the AWS cloud. The book will teach you about the most important services on AWS. You will also learn about best practices regarding security, high availability and scalability. You'll start with a broad overview of cloud computing and AWS and learn how to spin-up servers manually and from the command line. You'll learn how to automate your infrastructure by programmatically calling the AWS API to control every part of AWS. You will be introduced to the concept of Infrastructure as Code with the help of AWS CloudFormation. You will learn about different approaches to deploy applications on AWS. You'll also learn how to secure your infrastructure by isolating networks, controlling traffic and managing access to AWS resources. Next, you'll learn options and techniques for storing your data. You will experience how to integrate AWS services into your own applications by the use of SDKs. Finally, this book teaches you how to design for high availability, fault tolerance, and scalability.

What's inside

- Overview of AWS cloud concepts and best practices
- Manage servers on EC2 for cost-effectiveness
- Infrastructure automation with Infrastructure as Code (AWS CloudFormation)
- Deploy applications on AWS
- Store data on AWS: SQL, NoSQL, object storage and block storage
- Integrate Amazon's pre-built services
- Architect highly available and fault tolerant systems

Written for developers and DevOps engineers moving distributed applications to the AWS platform.

Docker in Practice

A *docker* was a labourer who was responsible for load goods of all sizes and shapes to a ship. Docker is also the name of a tool, helping you to deliver your applications to all kinds of machines. By using standardization, Docker allows you to automate packaging and deploying your applications. A nice side-effect is that this automation works on your local development machine as well as on cloud infrastructure. *Docker in Practice* helps you to get started with Docker. The first chapter, "Discovering Docker," explains the key concepts behind Docker.

Chapter 1 from *Docker in Practice* by Ian Miell and Aidan Hobson Sayers.

Discovering Docker

This chapter covers

- What Docker is
- The uses of Docker and how it can save you time and money
- The differences between containers and images
- Docker's layering feature
- Building and running a to-do application using Docker

Docker is a platform that allows you to "build, ship, and run any app, anywhere." It has come a long way in an incredibly short time and is now considered a standard way of solving one of the costliest aspects of software: deployment.

Before Docker came along, the development pipeline typically consisted of combinations of various technologies for managing the movement of software, such as virtual machines, configuration management tools, different package management systems, and complex webs of library dependencies. All these tools needed to be managed and maintained by specialist engineers, and most had their own unique ways of being configured.

Docker has changed all of this, allowing different engineers involved in this process to effectively speak one language, making working together a breeze. Everything goes through a common pipeline to a single output that can be used on any target—there's no need to continue maintaining a bewildering array of tool configurations, as shown in figure 1.1.

At the same time, there's no need to throw away your existing software stack if it works for you—you can package it up in a Docker container as-is for others to consume. As a bonus, you can see how these containers were built, so if you need to dig into the details, you can.

This book is aimed at intermediate developers with some knowledge of Docker. If you're OK with the basics, feel free to skip to the later chapters. The goal of this book is to expose the real-world challenges that Docker brings and show how they can be overcome. But first we're going to provide a quick refresher on Docker itself. If you want a more thorough treatment of Docker's basics, take a look at *Docker in Action* by Jeff Nickoloff (Manning Publications, 2016).

In chapter 2 you'll be introduced to Docker's architecture more deeply with the aid of some techniques that demonstrate its power. In this chapter you're going to learn what Docker is, see why it's important, and start using it.

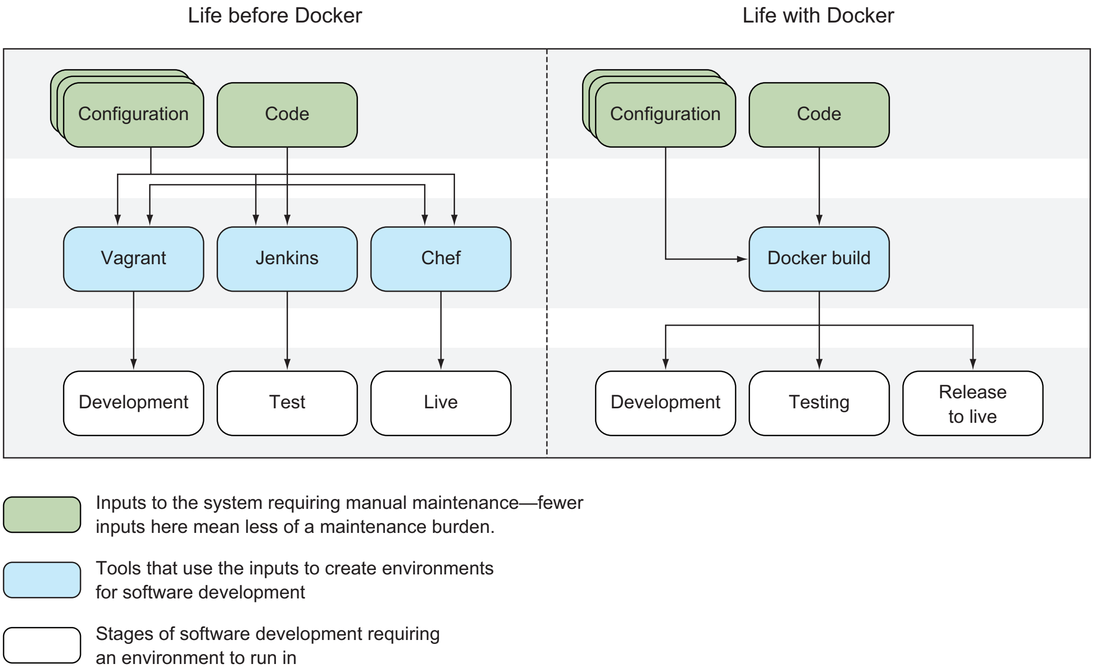

Life before Docker
Life with Docker
Configuration
Code
Vagrant
Jenkins
Chef
Development
Test
Live
Configuration
Code
Docker build
Development
Testing
Release to live
Inputs to the system requiring manual maintenance—fewer inputs here mean less of a maintenance burden.
Tools that use the inputs to create environments for software development
Stages of software development requiring an environment to run in

Figure 1.1 How Docker has eased the tool maintenance burden

1.1 The what and why of Docker

Before we get our hands dirty, we're going to discuss Docker a little so that you understand its context, where the name "Docker" came from, and why we're using it at all!

1.1.1 What is Docker?

To understand what Docker is, it's easier to start with a metaphor than a technical explanation, and the Docker metaphor is a powerful one. A docker was a labourer who moved commercial goods into and out of ships when they docked at ports. There were boxes and items of differing sizes and shapes, and experienced dockers were prized for their ability to fit goods into ships by hand in cost-effective ways (see figure 1.2). Hiring people to move stuff around wasn't cheap, but there was no alternative.

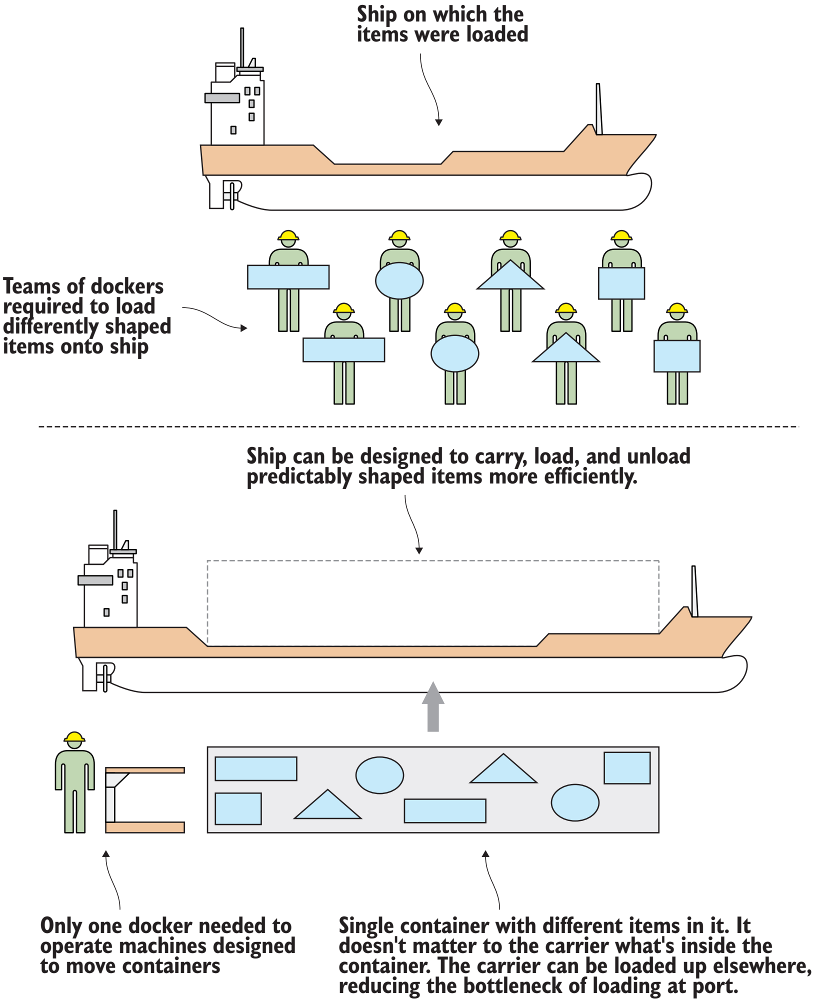

Ship on which the items were loaded
Teams of dockers required to load differently shaped items onto ship
Ship can be designed to carry, load, and unload predictably shaped items more efficiently.
Only one docker needed to operate machines designed to move containers
Single container with different items in it. It doesn't matter to the carrier what's inside the container. The carrier can be loaded up elsewhere, reducing the bottleneck of loading at port.

Figure 1.2 Shipping before and after standardized containers

This should sound familiar to anyone working in software. Much time and intellectual energy is spent getting metaphorically odd-shaped software into different sized metaphorical ships full of other odd-shaped software, so they can be sold to users or businesses elsewhere.

Figure 1.3 shows how time and money can be saved with the Docker concept.

Before Docker, deploying software to different environments required significant effort. Even if you weren't hand-running scripts to provision software on different machines (and plenty of people still do exactly that), you'd still have to wrestle with configuration management tools that manage state on what are increasingly fast-moving environments starved of resources. Even when these efforts were encapsulated in VMs, a lot of time was spent managing the deployment of these VMs, waiting for them to boot, and managing the overhead of resource use they created.

With Docker, the configuration effort is separated from the resource management, and the deployment effort is trivial: run `docker run`, and the environment's image is pulled down and ready to run, consuming fewer resources and contained so that it doesn't interfere with other environments.

You don't need to worry about whether your container is going to be shipped to a RedHat machine, an Ubuntu machine, or a CentOS VM image; as long as it has Docker on it, it'll be good to go.

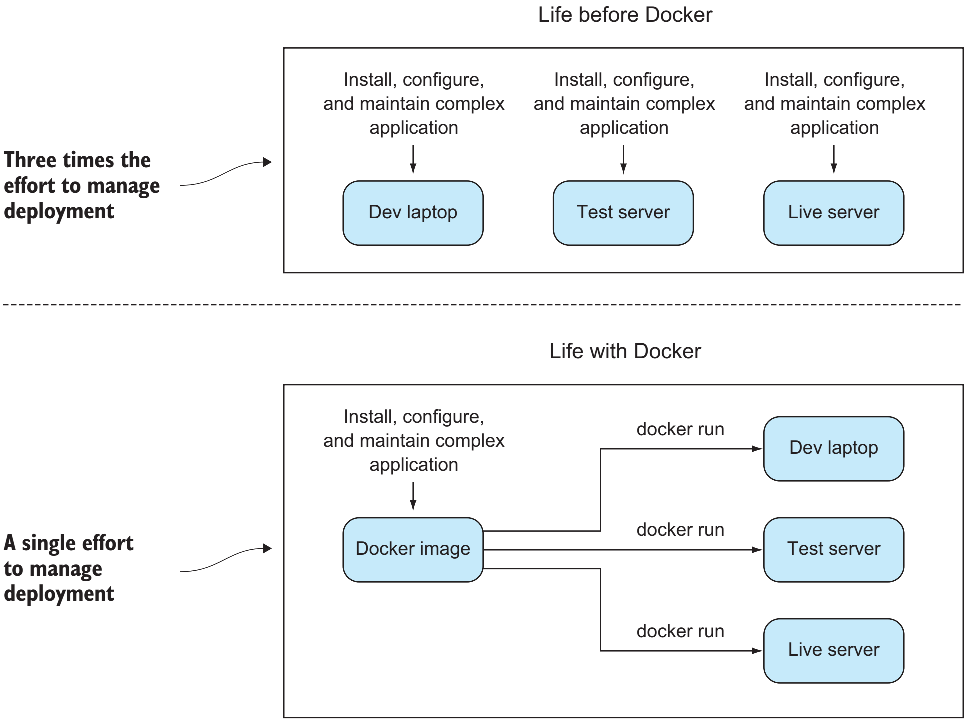

Life before Docker
Three times the effort to manage deployment
Install, configure, and maintain complex application
Install, configure, and maintain complex application
Install, configure, and maintain complex application
Dev laptop
Test server
Live server
Life with Docker
A single effort to manage deployment
Install, configure, and maintain complex application
Docker image
docker run
Dev laptop
docker run
Test server
docker run
Live server

Figure 1.3 Software delivery before and after Docker

1.1.2 What is Docker good for?

Some crucial practical questions arise: why would you use Docker, and for what? The short answer to the "why" is that for a modicum of effort, Docker can save your business a lot of money quickly. Some of these ways (and by no means all) are discussed in the following subsections. We've seen all of these benefits first-hand in real working contexts.

Replaces virtual machines (VMs)

Docker can be used to replace VMs in many situations. If you only care about the application, not the operating system, Docker can replace the VM, and you can leave worrying about the OS to someone else. Not only is Docker quicker than a VM to spin up, it's more lightweight to move around, and due to its layered filesystem, it's much easier and quicker to share changes with others. It's also firmly rooted in the command line and is eminently scriptable.

Prototyping software

If you want to quickly experiment with software without either disrupting your existing setup or going through the hassle of provisioning a VM, Docker can give you a sandbox environment in milliseconds. The liberating effect of this is difficult to grasp until you experience it for yourself.

Packaging software

Because a Docker image has effectively no dependencies for a Linux user, it's a great way to package software. You can build your image and be sure that it can run on any modern Linux machine—think Java, without the need for a JVM.

Enabling a microservices architecture

Docker facilitates the decomposition of a complex system to a series of composable parts, which allows you to reason about your services in a more discrete way. This can allow you to restructure your software to make its parts more manageable and pluggable without affecting the whole.

Modelling networks

Because you can spin up hundreds (even thousands) of isolated containers on one machine, modelling a network is a breeze. This can be great for testing real-world scenarios without breaking the bank.

Enabling full-stack productivity when offline

Because you can bundle all the parts of your system into Docker containers, you can orchestrate these to run on your laptop and work on the move, even when offline.

Reducing debugging overhead

Complex negotiations between different teams about software delivered is commonplace within the industry. We've personally experienced countless discussions about broken libraries; problematic dependencies; updates applied wrongly, or in the wrong order, or even not performed at all; unreproducible bugs, and so on. It's likely you have too. Docker allows you to state clearly (even in script form) the steps for debugging a problem on a system with known properties, making bug and environment reproduction a much simpler affair, and one normally separated from the host environment provided.

DOCUMENTING SOFTWARE DEPENDENCIES AND TOUCHPOINTS

By building your images in a structured way, ready to be moved to different environments, Docker forces you to document your software dependencies explicitly from a base starting point. Even if you decide not to use Docker everywhere, this need to document can help you install your software in other places.

ENABLING CONTINUOUS DELIVERY

Continuous delivery (CD) is a paradigm for software delivery based on a pipeline that rebuilds the system on every change and then delivers to production (or "live") through an automated (or partly automated) process.

Because you can control the build environment's state more exactly, Docker builds are more reproducible and replicable than traditional software building methods. This makes implementing CD much easier. Standard CD techniques such as Blue/Green deployment (where "live" and "last" deployments are maintained on live) and Phoenix Deployment (where whole systems are rebuilt on each release) are made trivial by implementing a reproducible Docker-centric build process.

Now you know a bit about how Docker can help you. Before we dive into a real example, let's go over a couple of core concepts.

1.1.3 Key concepts

In this section we're going to cover some key Docker concepts, which are illustrated in figure 1.4.

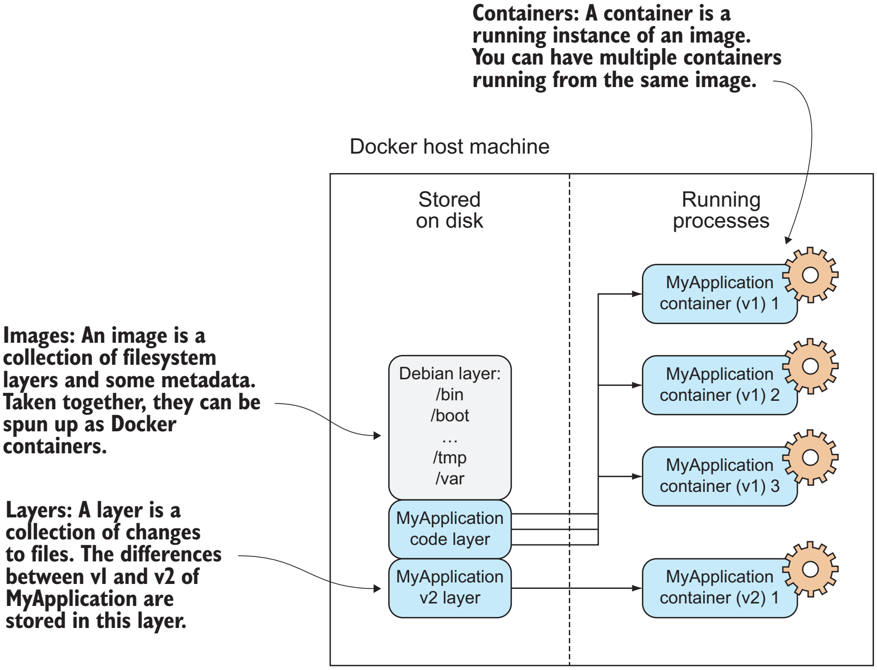

Containers: A container is a running instance of an image. You can have multiple containers running from the same image.
Docker host machine
Stored on disk
Running processes
MyApplication container (v1) 1
MyApplication container (v1) 2
MyApplication container (v1) 3
MyApplication container (v2) 1
Images: An image is a collection of filesystem layers and some metadata. Taken together, they can be spun up as Docker containers.
Debian layer: /bin /boot … /tmp /var
MyApplication code layer
MyApplication v2 layer
Layers: A layer is a collection of changes to files. The differences between vl and v2 of MyApplication are stored in this layer.

Figure 1.4 Core Docker concepts

It's most useful to get the concepts of images, containers, and layers clear in your mind before you start running Docker commands. In short, *containers* are running systems defined by *images*. These images are made up of one or more *layers* (or sets of diffs) plus some metadata for Docker.

Let's look at some of the core Docker commands. We'll turn images into containers, change them, and add layers to new images that we'll commit. Don't worry if all of this sounds confusing. By the end of the chapter it will all be much clearer!

Key Docker commands

Docker's central function is to build, ship, and run software in any location that has Docker.

To the end user, Docker is a command-line program that you run. Like git (or any source control tool), this program has subcommands that perform different operations.

The principal Docker subcommands you'll use on your host are listed in table 1.1.

Table 1.1 Docker subcommands

Command	Purpose
`docker build`	Build a Docker image.
`docker run`	Run a Docker image as a container.
`docker commit`	Commit a Docker container as an image.
`docker tag`	Tag a Docker image.

Images and containers

If you're unfamiliar with Docker, this may be the first time you've come across the words "container" and "image" in this context. They're probably the most important concepts in Docker, so it's worth spending a bit of time to make sure the difference is clear.

In figure 1.5 you'll see an illustration of these concepts, with three containers started up from one base image.

One way to look at images and containers is to see them as analogous to programs and processes. In the same way a process can be seen as an application being executed, a Docker container can be viewed as a Docker image in execution.

If you're familiar with object-oriented principles, another way to look at images and containers is to view images as classes and containers as objects. In the same way that objects are concrete instantiations of classes, containers are instantiations of images. You can create multiple containers from a single image, and they are all isolated from one another in the same way objects are. Whatever you change in the object, it won't affect the class definition—they're fundamentally different things.

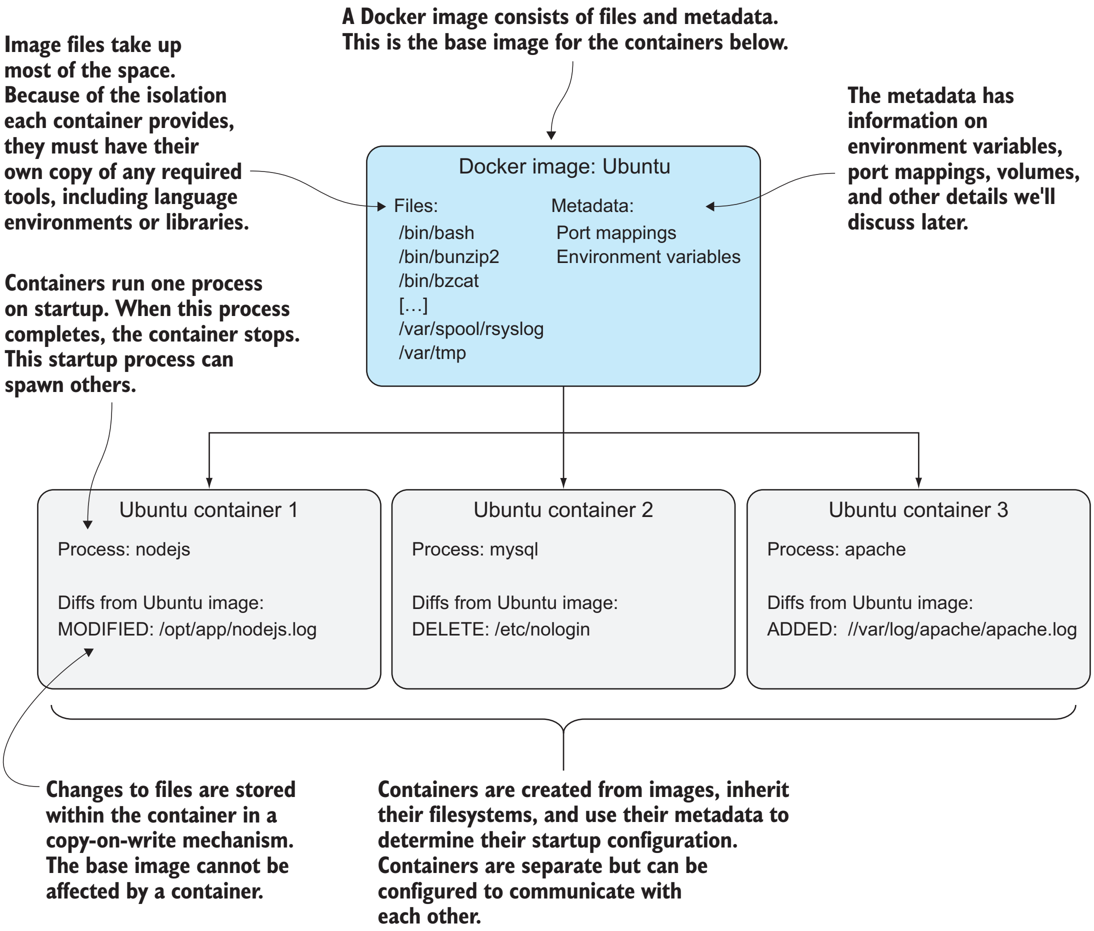

A Docker image consists of files and metadata. This is the base image for the containers below.
Image files take up most of the space. Because of the isolation each container provides, they must have their own copy of any required tools, including language environments or libraries.
The metadata has information on environment variables, port mappings, volumes, and other details we'll discuss later.
Docker image: Ubuntu
Files:
/bin/bash
/bin/bunzip2
/bin/bzcat
[…]
/var/spool/rsyslog
/var/tmp
Metadata:
Port mappings
Environment variables
Containers run one process on startup. When this process completes, the container stops. This startup process can spawn others.
Ubuntu container 1
Process: nodejs
Diffs from Ubuntu image:
MODIFIED: /opt/app/nodejs.log
Ubuntu container 2
Process: mysql
Diffs from Ubuntu image:
DELETE: /etc/nologin
Ubuntu container 3
Process: apache
Diffs from Ubuntu image:
ADDED: //var/log/apache/apache.log
Changes to files are stored within the container in a copy-on-write mechanism. The base image cannot be affected by a container.
Containers are created from images, inherit their filesystems, and use their metadata to determine their startup configuration. Containers are separate but can be configured to communicate with each other.

Figure 1.5 Docker images and containers

1.2 Building a Docker application

We're going to get our hands dirty now by building a simple "to-do" application (todoapp) image with Docker. In the process, you'll see some key Docker features like Dockerfiles, image re-use, port exposure, and build automation. Here's what you'll learn in the next 10 minutes:

- How to create a Docker image using a Dockerfile
- How to tag a Docker image for easy reference
- How to run your new Docker image

A to-do app is one that helps you keep track of things you want to get done. The app we'll build will store and display short strings of information that can be marked as done, presented in a simple web interface.

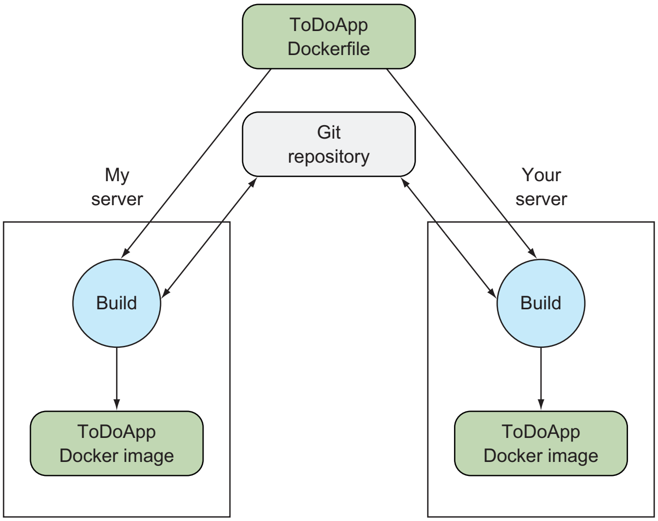


Figure 1.6 Building a Docker application

Figure 1.6 shows what we'll achieve by doing this.

The details of the application are unimportant. We're going to demonstrate that from the single short Dockerfile we're about to give you, you can reliably build, run, stop, and start an application in the same way on both your host and ours without needing to worry about application installations or dependencies. This is a key part of what Docker gives us—reliably reproduced and easily managed and shared development environments. This means no more complex or ambiguous installation instructions to follow and potentially get lost in.

THE TO-DO APPLICATION This to-do application will be used a few times throughout the book, and it's quite a useful one to play with and demonstrate, so it's worth familiarizing yourself with it.

1.2.1 Ways to create a new Docker image

There are four standard ways to create Docker images. Table 1.2 itemizes these methods.

Table 1.2 Options for creating Docker images

Method	Description	See technique
Docker commands / "By hand"	Fire up a container with `docker run` and input the commands to create your image on the command line. Create a new image with `docker commit`.	See technique 14.
Dockerfile	Build from a known base image, and specify build with a limited set of simple commands.	Discussed shortly.

Table 1.2 Options for creating Docker images *(continued)*

Method	Description	See technique
Dockerfile and configuration management (CM) tool	Same as Dockerfile, but hand over control of the build to a more sophisticated CM tool.	See technique 47.
Scratch image and import a set of files	From an empty image, import a TAR file with the required files.	See technique 10.

The first "by hand" option is fine if you're doing proofs of concept to see whether your installation process works. At the same time, you should be keeping notes about the steps you're taking so that you can return to the same point if you need to.

At some point you're going to want to define the steps for creating your image. This is the second option (and the one we'll use here).

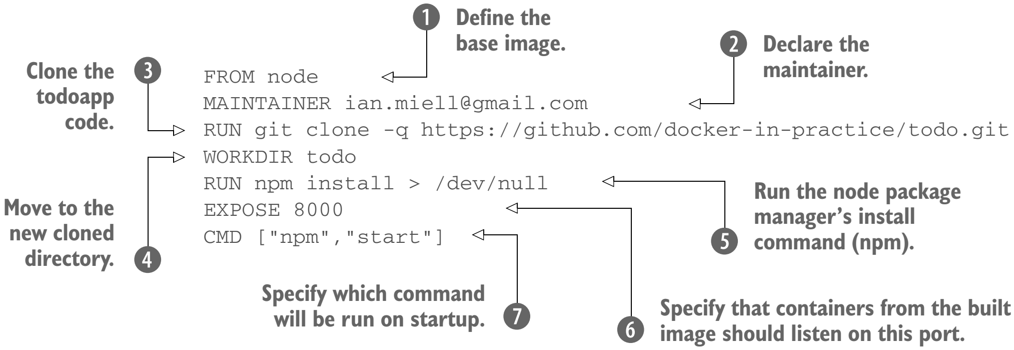
For more complex builds, you may want to go for the third option, particularly when the Dockerfile features aren't sophisticated enough for your image's needs.

The final option builds from a null image by overlaying the set of files required to run the image. This is useful if you want to import a set of self-contained files created elsewhere, but it's rarely seen in mainstream use.

We'll look at the Dockerfile method now; the other methods will be covered later in the book.

1.2.2 Writing a Dockerfile

A Dockerfile is a text file with a series of commands in it. Here's the Dockerfile we're going to use for this example:



```
FROM node
MAINTAINER ian.miell@gmail.com
RUN git clone -q https://github.com/docker-in-practice/todo.git
WORKDIR todo
RUN npm install > /dev/null
EXPOSE 8000
CMD ["npm","start"]
```

❶ Define the base image.

❷ Declare the maintainer.

❸ Clone the todoapp code.

❹ Move to the new cloned directory.

❺ Run the node package manager's install command (npm).

❻ Specify that containers from the built image should listen on this port.

❼ Specify which command will be run on startup.

You begin the Dockerfile by defining the base image with the FROM command ❶. This example uses a Node.js image so you have access to the Node.js binaries. The official Node.js image is called node.

Next, you declare the maintainer with the MAINTAINER command ❷. In this case, we're using one of our email addresses, but you can replace this with your own reference because it's your Dockerfile now. This line isn't required to make a working Docker image, but it's good practice to include one. At this point, the build has inherited the state of the node container, and you're ready to work on top of it.

Next, you clone the todoapp code with a `RUN` command ❸. This uses the specified command to retrieve the code for the application, running `git` within the container. Git is installed inside the base node image in this case, but you can't take this kind of thing for granted.

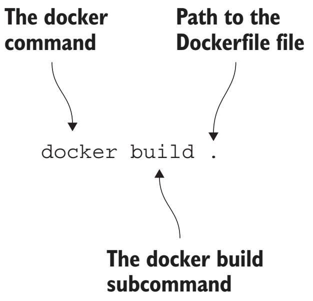
Now you move to the new cloned directory with a `WORKDIR` command ❹. Not only does this change directory within the build context, but the last `WORKDIR` command determines which directory you're in by default when you start up your container from your built image.

Next, you run the node package manager's install command (`npm`) ❺. This will set up the dependencies for your application. You aren't interested in the output here, so you redirect it to /dev/null.

Because port 8000 is used by the application, you use the `EXPOSE` command to tell Docker that containers from the built image should listen on this port ❻.

Finally, you use the `CMD` command to tell Docker which command will be run on startup of the container ❼.

This simple example illustrates several key features of Docker and Dockerfiles. A Dockerfile is a simple sequence of a limited set of commands run in strict order. They affect the files and metadata of the resulting image. Here the `RUN` command affects the filesystem by checking out and installing applications, and the `EXPOSE`, `CMD`, and `WORKDIR` commands affect the metadata of the image.



```
docker build .
```

Figure 1.7 Docker `build` command

1.2.3 Building a Docker image

You've defined your Dockerfile's build steps. Now you're going to build the Docker image from it by typing the command in figure 1.7.

The output you'll see will be similar to this:

```
Sending build context to Docker daemon 178.7 kB
Sending build context to Docker daemon
Step 0 : FROM node
 ---> fc81e574af43
Step 1 : MAINTAINER ian.miell@gmail.com
 ---> Running in 21af1aad6950
 ---> 8f32669fe435
Removing intermediate container 21af1aad6950
Step 2 : RUN git clone https://github.com/ianmiell/todo.git
 ---> Running in 0a030ee746ea
Cloning into 'todo'...
```

Docker uploads the files and directories under the path supplied to the docker build command.

Each build step is numbered sequentially from 0 and output with the command.

Each command results in a new image being created, and the image ID is output.

To save space, each intermediate container is removed before continuing.

```
 ---> 783c68b2e3fc
Removing intermediate container 0a030ee746ea
Step 3 : WORKDIR todo
 ---> Running in 2e59f5df7152
 ---> 8686b344b124
Removing intermediate container 2e59f5df7152
Step 4 : RUN npm install
 ---> Running in bdf07a308fca
npm info it worked if it ends with ok
[...]
npm info ok
 ---> 6cf8f3633306
Removing intermediate container bdf07a308fca
Step 5 : RUN chmod -R 777 /todo
 ---> Running in c03f27789768
 ---> 2c0ededd3a5e
Removing intermediate container c03f27789768
Step 6 : EXPOSE 8000
 ---> Running in 46685ea97b8f
 ---> f1c29feca036
Removing intermediate container 46685ea97b8f
Step 7 : CMD npm start
 ---> Running in 7b4c1a9ed6af
 ---> 439b172f994e
Removing intermediate container 7b4c1a9ed6af
Successfully built 439b172f994e
```

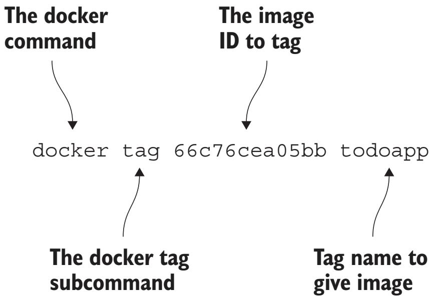
Debug of the build is output here (and edited out of this listing).

Final image ID for this build, ready to tag

You now have a Docker image with an image ID ("66c76cea05bb" in the preceding example, but your ID will be different). It can be cumbersome to keep referring to this ID, so you can tag it for easier reference.

Type the preceding command, replacing the 66c76cea05bb with whatever image ID was generated for you.

You can now build your own copy of a Docker image from a Dockerfile, reproducing an environment defined by someone else!



```
docker tag 66c76cea05bb todoapp
```

Figure 1.8 Docker `tag` command

1.2.4 Running a Docker container

You've built and tagged your Docker image. Now you can run it as a container:

```
docker run -p 8000:8000 --name example1 todoapp
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm info prestart todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 prestart /todo
> make all
```

1 The docker run subcommand starts the container, -p maps the container's port 8000 to the port 8000 on the host machine, --name gives the container a unique name, and the last argument is the image.

The output of the container's starting process is sent to the terminal.

```
npm install
npm info it worked if it ends with ok
npm info using npm@2.14.4
npm info using node@v4.1.1
npm WARN package.json todomvc-swarm@0.0.1 No repository field.
npm WARN package.json todomvc-swarm@0.0.1 license should be a
➥ valid SPDX license expression
npm info preinstall todomvc-swarm@0.0.1
npm info package.json statics@0.1.0 license should be a valid
➥ SPDX license expression
npm info package.json react-tools@0.11.2 No license field.
npm info package.json react@0.11.2 No license field.
npm info package.json node-jsx@0.11.0 license should be a valid
➥ SPDX license expression
npm info package.json ws@0.4.32 No license field.
npm info build /todo
npm info linkStuff todomvc-swarm@0.0.1
npm info install todomvc-swarm@0.0.1
npm info postinstall todomvc-swarm@0.0.1
npm info prepublish todomvc-swarm@0.0.1
npm info ok
if [ ! -e dist/ ]; then mkdir dist; fi
cp node_modules/react/dist/react.min.js dist/react.min.js

LocalTodoApp.js:9:    // TODO: default english version
LocalTodoApp.js:84:            fwdList =
➥ this.host.get('/TodoList#'+listId); // TODO fn+id sig
TodoApp.js:117:        // TODO scroll into view
TodoApp.js:176:        if (i>=list.length()) { i=list.length()-1; }
➥ // TODO .length
local.html:30:    <!-- TODO 2-split, 3-split -->
model/TodoList.js:29:
➥ // TODO one op - repeated spec? long spec?
view/Footer.jsx:61:        // TODO: show the entry's metadata
view/Footer.jsx:80:            todoList.addObject(new TodoItem());
➥ // TODO create default
view/Header.jsx:25:
➥ // TODO list some meaningful header (apart from the id)

npm info start todomvc-swarm@0.0.1

> todomvc-swarm@0.0.1 start /todo
> node TodoAppServer.js

Swarm server started port 8000
^C
$ docker ps -a
CONTAINER ID  IMAGE           COMMAND      CREATED
➥ STATUS                        PORTS  NAMES
b9db5ada0461  todoapp:latest  "npm start"  2 minutes ago
➥ Exited (130) 2 minutes ago    example1
$ docker start example1
```

2 Hit Ctrl-C here to terminate the process and the container.

3 Run this command to see containers that have been started and removed, along with an ID and status (like a process).

4 Restart the container, this time in the background.

Run the ps command again to see the changed status. ❺

```
example1
$ docker ps -a
CONTAINER ID   IMAGE            COMMAND        CREATED
➥ STATUS          PORTS                        NAMES
b9db5ada0461   todoapp:latest   "npm start"    8 minutes ago
➥ Up 10 seconds   0.0.0.0:8000->8000/tcp    example1
$ docker diff example1
C /todo
A /todo/.swarm
A /todo/.swarm/TodoItem
A /todo/.swarm/TodoItem/1tlOc02+A~4UZcz
A /todo/.swarm/_log
A /todo/dist
A /todo/dist/LocalTodoApp.app.js
A /todo/dist/TodoApp.app.js
A /todo/dist/react.min.js
```

The docker diff subcommand shows you what files have been affected since the image was instantiated as a container. ❻

❼ The /todo directory has been changed.

❽ The /todo/.swarm directory has been added.

The `docker run` subcommand starts up the container ❶. The -p flag maps the container's port 8000 to the port 8000 on the host machine, so you should now be able to navigate with your browser to http://localhost:8000 to view the application. The --name flag gives the container a unique name you can refer to later for convenience. The last argument is the image name.

Once the container was started, we hit CTRL-C to terminate the process and the container ❷. You can run the `ps` command to see the containers that have been started but not removed ❸. Note that each container has its own container ID and status, analogous to a process. Its status is `Exited`, but you can restart it ❹. After you do, notice how the status has changed to `Up` and the port mapping from container to host machine is now displayed ❺.

The `docker diff` subcommand shows you which files have been affected since the image was instantiated as a container ❻. In this case, the todo directory has been changed ❼ and the other listed files have been added ❽. No files have been deleted, which is the other possibility.

As you can see, the fact that Docker "contains" your environment means that you can treat it as an entity on which actions can be predictably performed. This gives Docker its breadth of power—you can affect the software lifecycle from development to production and maintenance. These changes are what this book will cover, showing you in practical terms what can be done with Docker.

Next you're going to learn about layering, another key concept in Docker.

1.2.5 Docker layering

Docker layering helps you manage a big problem that arises when you use containers at scale. Imagine what would happen if you started up hundreds—or even thousands—of the to-do app, and each of those required a copy of the files to be stored somewhere.

As you can imagine, disk space would run out pretty quickly! By default, Docker internally uses a copy-on-write mechanism to reduce the amount of disk space required

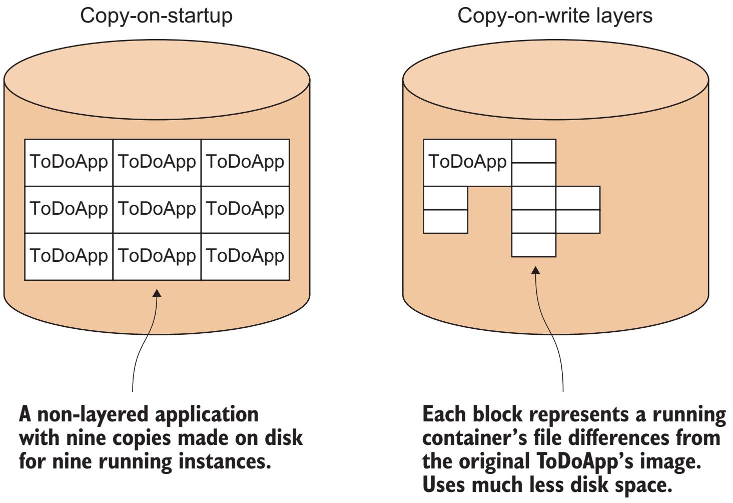

Copy-on-startup
Copy-on-write layers
ToDoApp ToDoApp ToDoApp
ToDoApp ToDoApp ToDoApp
ToDoApp ToDoApp ToDoApp
ToDoApp
A non-layered application with nine copies made on disk for nine running instances.
Each block represents a running container's file differences from the original ToDoApp's image. Uses much less disk space.

Figure 1.9 Copy-on-startup vs copy-on-write

(see figure 1.9). Whenever a running container needs to write to a file, it records the change by copying the item to a new area of disk. When a Docker commit is performed, this new area of disk is frozen and recorded as a layer with its own identifier.

This partly explains how Docker containers can start up so quickly—they have nothing to copy because all the data has already been stored as the image.

COPY-ON-WRITE Copy-on-write is a standard optimization strategy used in computing. When you create a new object (of any type) from a template, rather than copying the entire set of data required, you only copy data over when it's changed. Depending on the use case, this can save considerable resources.

Figure 1.10 illustrates that the to-do app you've built has three layers you're interested in.

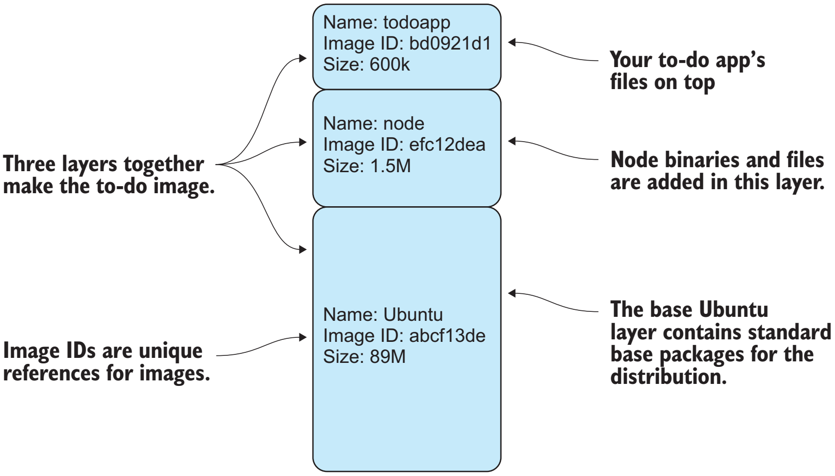

Name: todoapp
Image ID: bd0921d1
Size: 600k
Name: node
Image ID: efc12dea
Size: 1.5M
Name: Ubuntu
Image ID: abcf13de
Size: 89M
Three layers together make the to-do image.
Image IDs are unique references for images.
Your to-do app's files on top
Node binaries and files are added in this layer.
The base Ubuntu layer contains standard base packages for the distribution.

Figure 1.10 The to-do app's filesystem layering in Docker

Because the layers are static, you only need build on top of the image you wish to take as a reference, should you need anything to change in a higher layer. In the to-do app, you built from the publicly available node image and layered changes on top.

All three layers can be shared across multiple running containers, much as a shared library can be shared in memory across multiple running processes. This is a vital feature for operations, allowing the running of numerous containers based on different images on host machines without running out of disk space.

Imagine that you're running the to-do app as a live service for paying customers. You can scale up your offering to a large number of users. If you're developing, you can spin up many different environments on your local machine at once. If you're moving through tests, you can run many more tests simultaneously, and far more quickly than before. All these things are made possible by layering.

By building and running an application with Docker, you've begun to see the power that Docker can bring to your workflow. Reproducing and sharing specific environments and being able to land these in various places gives you both flexibility and control over development.

1.3 Summary

Depending on your previous experience with Docker, this chapter might have been a steep learning curve. We've covered a lot of ground in a short time.

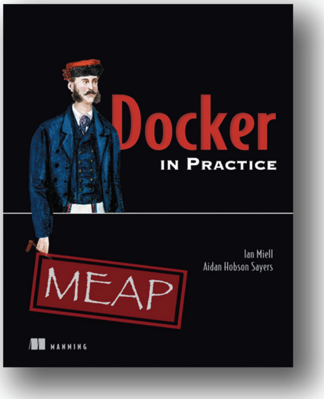
You should now

- Understand what a Docker image is
- Know what Docker layering is, and why it's useful
- Be able to commit a new Docker image from a base image
- Know what a Dockerfile is

We've used this knowledge to

- Create a useful application
- Reproduce state in an application with minimal effort

Next we're going to introduce techniques that will help you understand how Docker works and, from there, discuss some of the broader technical debate around Docker's usage. These first two introductory chapters form the basis for the remainder of the book, which will take you from development to production, showing you how Docker can be used to improve your workflow.



Docker is impossible to ignore. This lightweight container system is easier to deploy and more flexible than traditional VMs. Built for simplicity and speed, it radically reduces your reliance on manual system administration for tasks like configuring servers, creating disposable (and portable!) development environments, and predictably rolling out applications on unknown systems. While the idea behind Docker is simple, it can have a major impact on how you develop and deploy software.

Docker in Practice is a hands-on guide to over 100 specific techniques you can use to get the most out of Docker in your daily work. Following a cookbook-style Problem/Solution/Discussion format, this practical handbook gives you instantly-useful solutions for important areas like effortless server maintenance and configuration, deploying microservices, creating safe environments for experimentation, and much more. As you read, you'll graduate from Docker basics into must-have practices like integrating Docker with your Continuous Integration process, automating complex container creation with Chef, and orchestration with Kubernetes.

What's inside

- Speed up your DevOps pipeline with the use of containers
- Reduce the effort of maintaining and configuring software
- Using Docker to cheaply replace VMs
- Streamlining your cloud workflow
- Using the Docker Hub and its workflow
- Navigating the Docker ecosystem

Written for developers and devops engineers who have already started their Docker journey and want to use it effectively in a production setting.

Mesos in Action

Managing a growing cloud infrastructure can become a complex task especially if you need to operate using multiple cloud providers as well as on-site environments. Mesos adds a layer of abstraction on top of your infrastructure. This means you are able to manage whole infrastructures as if everything is running on a single machine. *Mesos in Action* introduces you to the Apache Mesos cluster manager and the concept of application-centric infrastructure. Chapter 1, "Introducing Mesos," gives you a look behind the scenes of the abstraction layer.

Chapter 1 from *Mesos in Action* by Roger Ignazio.

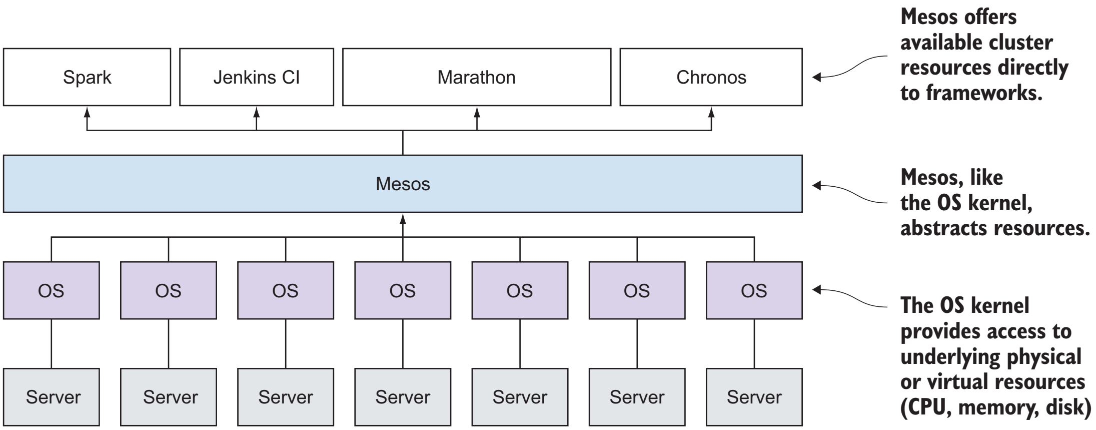
Introducing Mesos

This chapter covers

- Introducing Mesos
- Comparing Mesos with a traditional datacenter
- Understanding when and why to use Mesos
- Working with Mesos's distributed architecture

Traditionally, physical—and virtual—machines have been the typical units of computing in a datacenter. Machines are provisioned with various configuration management tools to later have applications deployed. These machines are usually organized into clusters providing individual services, and systems administrators oversee their day-to-day operations. Eventually, these clusters reach their maximum capacity, and more machines are brought online to handle the load.

In 2010, a project at the University of California, Berkeley, aimed to solve the scaling problem. The software project, now known as *Apache Mesos*, abstracts CPU, memory, and disk resources in a way that allows datacenters to function as if they were one large machine. Mesos creates a single underlying cluster to provide applications with the resources they need, without the overhead of virtual machines and operating systems. You can see a simplified example of this in figure 1.1.



Mesos offers available cluster resources directly to frameworks.

Mesos, like the OS kernel, abstracts resources.

The OS kernel provides access to underlying physical or virtual resources (CPU, memory, disk)

Figure 1.1 Frameworks sharing datacenter resources offered by Mesos

This book introduces Apache Mesos, an open source cluster manager that allows systems administrators and developers to focus less on individual servers and more on the applications that run on them. You'll see how to get up and running with Mesos in your environment, how it shares resources and handles failure, and—perhaps most important—how to use it as a platform to deploy applications.

1.1 Meet Mesos

Mesos works by introducing a layer of abstraction that provides a means to use entire datacenters as if they were a single, large server. Instead of focusing on one application running on a specific server, Mesos's resource isolation allows for multitenancy—the ability to run multiple applications on a single machine—leading to more efficient use of computing resources.

To better understand this concept, you might think of Mesos as being similar to today's virtualization solutions: just as a hypervisor abstracts physical CPU, memory, and storage resources and presents them to virtual machines, Mesos does the same but offers these resources directly to applications. Another way to think about this is in the context of multicore processors: when you launch an application on your laptop, it runs on one or more cores, but in most cases it doesn't particularly matter which one. Mesos applies this same concept to datacenters.

In addition to improving overall resource use, Mesos is distributed, highly available, and fault-tolerant right out of the box. It has built-in support for isolating processes using containers, such as Linux control groups (cgroups) and Docker, allowing multiple applications to run alongside each other on a single machine. Where you once might have set up three clusters—one each to run Memcached, Jenkins CI, and your Ruby on Rails apps—you can instead deploy a single Mesos cluster to run all of these applications.

In the next few sections, you're going to look at how Mesos works to provide all of these features and how it compares to a traditional datacenter.

1.1.1 Understanding how it works

Using a combination of concepts referred to as resource offers, two-tier scheduling, and resource isolation, Mesos provides a means for the cluster to act as a single supercomputer on which to run tasks. Before digging in too deeply here, let's take a look at figure 1.2. This diagram demonstrates the logic Mesos follows when offering resources to running applications. This particular example references the Apache Spark data-processing framework.

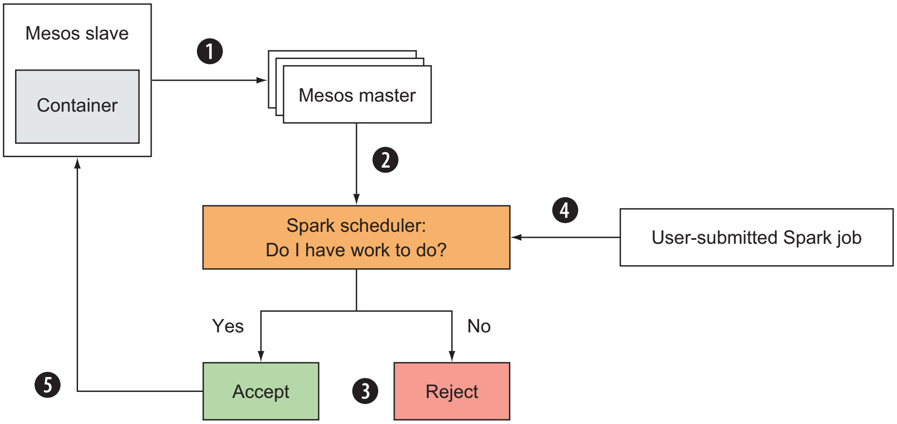


Figure 1.2 Mesos advertises the available CPU, memory, and disk as resource offers to frameworks.

Let's break it down:

1. The Mesos slave offers its available CPU, memory, and disk to the Mesos *master* in the form of a *resource offer.*
2. The Mesos master's *allocation module*—or scheduling algorithm—decides which *frameworks*—or applications—to offer the resources to.
3. In this particular case, the Spark *scheduler* doesn't have any jobs to run on the cluster. It rejects the resource offer, allowing the master to offer the resources to another framework that might have some work to do.
4. Now consider a user submitting a Spark job to be run on the cluster. The scheduler accepts the job and waits for a resource offer that satisfies the workload.
5. The Spark scheduler accepts a resource offer from the Mesos master, and launches one or more *tasks* on an available Mesos *slave.* These tasks are launched

within a container, providing isolation between the various tasks that might be running on a given Mesos slave.

Seems simple, right? Now that you've learned how Mesos uses resource offers to advertise resources to frameworks, and how two-tier scheduling allows frameworks to accept and reject resource offers as needed, let's take a closer look at some of these fundamental concepts.

> **NOTE** An effort is underway to rename the Mesos *slave* role to *agent* for future versions of Mesos. Because this book covers Mesos 0.22.2, it uses the terminology of that specific release, so as to not create any unnecessary confusion. For more information, see https://issues.apache.org/jira/browse/MESOS-1478.

Resource Offers

Like many other cluster managers, Mesos clusters are made up of groups of machines called *masters* and *slaves*. Each Mesos slave in a cluster advertises its available CPU, memory, and storage in the form of resource offers. As you saw in figure 1.2, these resource offers are periodically sent from the slaves to the Mesos masters, processed by a scheduling algorithm, and then offered to a framework's scheduler running on the Mesos cluster.

Two-tier Scheduling

In a Mesos cluster, resource scheduling is the responsibility of the Mesos master's allocation module and the framework's scheduler, a concept known as *two-tier scheduling*. As previously demonstrated, resource offers from Mesos slaves are sent to the master's allocation module, which is then responsible for offering resources to various framework schedulers. The framework schedulers can accept or reject the resources based on their workload.

The allocation module is a pluggable component of the Mesos master that implements an algorithm to determine which offers are sent to which frameworks (and when). The modular nature of this component allows systems engineers to implement their own resource-sharing policies for their organization. By default, Mesos uses an algorithm developed at UC Berkeley known as Dominant Resource Fairness (DRF):

> *In a nutshell, DRF seeks to maximize the minimum dominant share across all users. For example, if user A runs CPU-heavy tasks and user B runs memory-heavy tasks, DRF attempts to equalize user A's share of CPUs with user B's share of memory. In the single-resource case, DRF reduces to max-min fairness for that resource.*[1]

[1] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." NSDI, vol. 11, 2011.

Mesos's use of the DRF algorithm by default is fine for most deployments. Chances are you won't need to write your own allocation algorithm, so this book doesn't go into much detail about DRF. If you're interested in learning more about this research, you can find the paper online at www.usenix.org/legacy/events/nsdi11/tech/full_papers/Ghodsi.pdf.

RESOURCE ISOLATION

Using Linux cgroups or Docker containers to isolate processes, Mesos allows for *multitenancy*, or for multiple processes to be executed on a single Mesos slave. A framework then executes its tasks within the container, using a Mesos *containerizer*. If you're not familiar with containers, think of them as a lightweight approach to how a hypervisor runs multiple virtual machines on a single physical host, but without the overhead or need to run an entire operating system.

> **NOTE** In addition to Docker and cgroups, Mesos provides another means of isolation for other POSIX-compliant operating systems: `posix/cpu`, `posix/mem`, and `posix/disk`. It's worth noting that these isolation methods don't *isolate* resources, but instead monitor resource use.

Now that you have a clearer understanding of how Mesos works, you can move on to understanding how this technology compares to the traditional datacenter. More specifically, the next section introduces the concept of an application-centric datacenter, where the focus is more on applications than on the servers and operating systems that run them.

1.1.2 Comparing virtual machines and containers

When thinking about applications deployed in a traditional datacenter, virtual machines often come to mind. In recent years, virtualization providers (VMware, OpenStack, Xen, and KVM, to name a few) have become commonplace in many organizations. Similar to how a hypervisor allows a physical host's resources to be abstracted and shared among virtual machines, Mesos provides a layer of abstraction, albeit at a different level. The resources are presented to applications themselves, and in turn consumed by containers.

To illustrate this point, consider figure 1.3, which compares the various layers of infrastructure required to deploy four applications.

VIRTUAL MACHINES

When thinking about traditional virtual machine–based application deployments, consider for a moment the operational overhead of maintaining the operating systems on each of them: installing packages, applying security updates, maintaining user access, identifying and remediating configuration drift; the list goes on. What's the added benefit of running applications atop an entire operating system when you're more concerned with deploying the application itself? Not to mention the

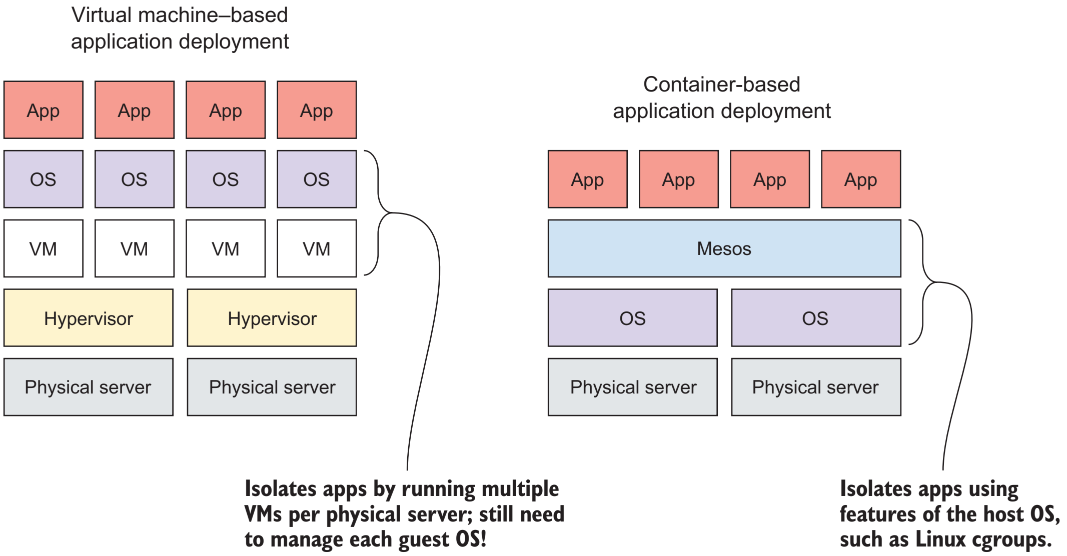

Virtual machine–based application deployment
App
App
App
App
OS
OS
OS
OS
VM
VM
VM
VM
Hypervisor
Hypervisor
Physical server
Physical server
Container-based application deployment
App
App
App
App
Mesos
OS
OS
Physical server
Physical server
Isolates apps by running multiple VMs per physical server; still need to manage each guest OS!
Isolates apps using features of the host OS, such as Linux cgroups.

Figure 1.3 Comparing VM-based and container-based application deployments

overhead of the operating system, which consumes added CPU, memory, and disk. At a large-enough scale, this becomes wasteful. With an application-centric approach to managing datacenters, Mesos allows you to simplify your stack—and your application deployments—using lightweight containers.

CONTAINERS

As you learned previously, Mesos uses containers for resource isolation between processes. In the context of Mesos, the two most important resource-isolation methods to know about are the control groups (cgroups) built into the Linux kernel, and Docker.

Around 2007, support for control groups (referred to as *cgroups* throughout this text) was made available in the Linux kernel, beginning with version 2.6.24. This allows the execution of processes in a way that's *sandboxed* from other processes. In the context of Mesos, cgroups provide resource constraints for running processes, ensuring that they don't interfere with other processes running on the system. When using cgroups, any packages or libraries that the tasks might depend on (a specific version of Python, a working C++ compiler, and so on) must be already present on the host operating system. If your workloads, packages, and required tools and libraries are fairly standardized or don't readily conflict with each other, this might not be a problem. But consider figure 1.4, which demonstrates how using Docker can overcome these sorts of problems and allow you to run applications and workloads in a more isolated manner.

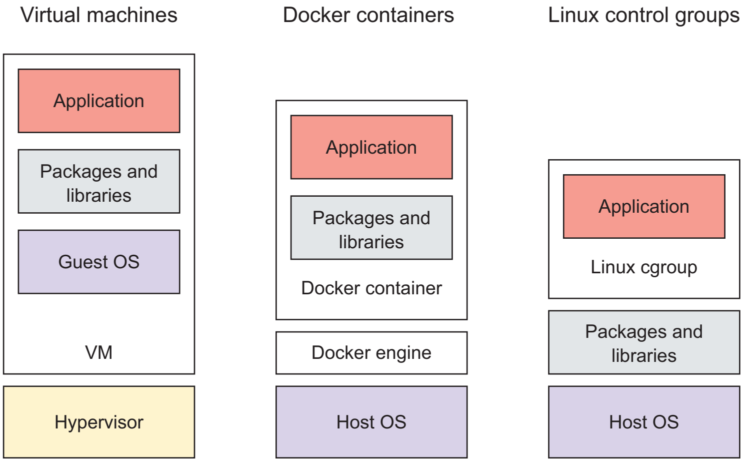


Figure 1.4 Comparing virtual machines, Docker containers, and Linux cgroups

Using low-level primitives in the Linux kernel, including cgroups and namespaces, Docker provides a means to build and deploy containers almost as if they were virtual machines. The application and all of its dependencies are packaged within the container and deployed atop a host operating system. They take a concept from the freight industry—the standardized industrial shipping container—and apply this to application deployment. In recent years, this new unit of software delivery has grown in popularity as it's generally considered to be more lightweight than deploying an entire virtual machine.

You don't need to understand all the implementation details and intricacies of building and deploying containers to use Mesos, though. If you'd like more information, please consult the following online resources:

- Linux control groups: www.kernel.org/doc/documentation/cgroup-v1/cgroups.txt
- Docker: https://docs.docker.com

1.1.3 Knowing when (and why) to use Mesos

Running applications at scale isn't reserved for large enterprises anymore. Startups with only a handful of employees are creating apps that easily attract millions of users. Re-architecting applications and datacenters is a nontrivial task, but certain components that are in a typical stack are already great candidates to run on Mesos. By taking some of these technologies and moving them (and their workloads) to a Mesos cluster, you can scale them more easily and run your datacenter more efficiently.

NOTE This book covers Mesos version 0.22.2, which provides an environment for running stateless and distributed applications. Beginning in version 0.23, Mesos will begin work to support persistent resources, thus enabling support for stateful frameworks. For more information on this effort, see https://issues.apache.org/jira/browse/MESOS-1554.

For example, consider the stateless, distributed, and stateful technologies in table 1.1.

Table 1.1 Technologies that are—and aren't—good candidates to run on Mesos

Service type	Examples	Should you use Mesos?
Stateless—no need to persist data to disk	Web apps (Ruby on Rails, Play, Django), Memcached, Jenkins CI build slaves	Yes
Distributed out of the box	Cassandra, Elasticsearch, Hadoop Distributed File System (HDFS)	Yes, provided the correct level of redundancy is in place
Stateful—needs to persist data to disk	MySQL, PostgreSQL, Jenkins CI masters	No (version 0.22); potentially (version 0.23+)

The real value of Mesos is realized when running stateless services and applications—applications that will handle incoming loads but that could go offline at any time without negatively impacting the service as a whole, or services that run a job and report the result to another system. As noted previously, examples of some of these applications include Ruby on Rails and Jenkins CI build slaves.

Progress has been made running distributed databases (such as Cassandra and Elasticsearch) and distributed filesystems (such as Hadoop Distributed File System, or HDFS) as Mesos frameworks. But this is feasible only if the correct level of redundancy is in place. Although certain distributed databases and filesystems have data replication and fault tolerance built in, your data might not survive if the entire Mesos cluster fails (because of natural disasters, redundant power/cooling systems failures, or human error). In the real world, you should weigh the risks and benefits of deploying services that persist data on a Mesos cluster.

As I mentioned earlier, Mesos excels at running stateless, distributed services. Stateful applications that need to persist data to disk aren't good candidates for running on Mesos as of this writing. Although possible, it's not yet advisable to run certain databases such as MySQL and PostgreSQL atop a Mesos cluster. When you do need to persist data, it's preferable to do so by deploying a traditional database cluster outside the Mesos cluster.

1.2 Why we need to rethink the datacenter

Deploying applications within a datacenter has traditionally involved one or more physical (or virtual) servers. The introduction and mainstream adoption of virtualization has allowed us to run multiple virtual machines on a single physical server and make better use of physical resources. But running applications this way also means you're usually running a full operating system on each of those virtual machines, which consumes resources and brings along its own maintenance overhead.

This section presents two primary reasons that you should rethink how datacenters are managed: the administrative overhead of statically partitioning resources, and the need to focus more on applications instead of infrastructure.

1.2.1 Partitioning of resources

When you consider the traditional virtual machine–based model of deploying applications and statically partitioning clusters, you quickly find this deployment model inefficient and cumbersome to maintain. By maximizing the use of each server in a datacenter, operations teams maximize their return on investment and can keep the total cost of ownership as reasonable as possible.

In computing circles, teams generally refer to a *cluster* as a group of servers that work together as a single system to provide a service. Traditionally, the deployment of these services has been largely node-centric: you dedicate a certain number of machines to provide a given service. But as the infrastructure footprint expands and service offerings increase, it's difficult to continue statically partitioning these services.

But now consider the demand for these services doubling. To continue scaling, a systems administrator needs to provision new machines and join them to the individual clusters. Perhaps the operations team, anticipating the need for additional capacity, scales each of those clusters to three times its current size. Although you've managed to scale each of those services, you now have machines in your datacenter sitting idle, waiting to be used. As such, if a single machine in any of those clusters fails, it quickly needs to be brought back online for the service to continue operating at full capacity, as shown in figure 1.5.

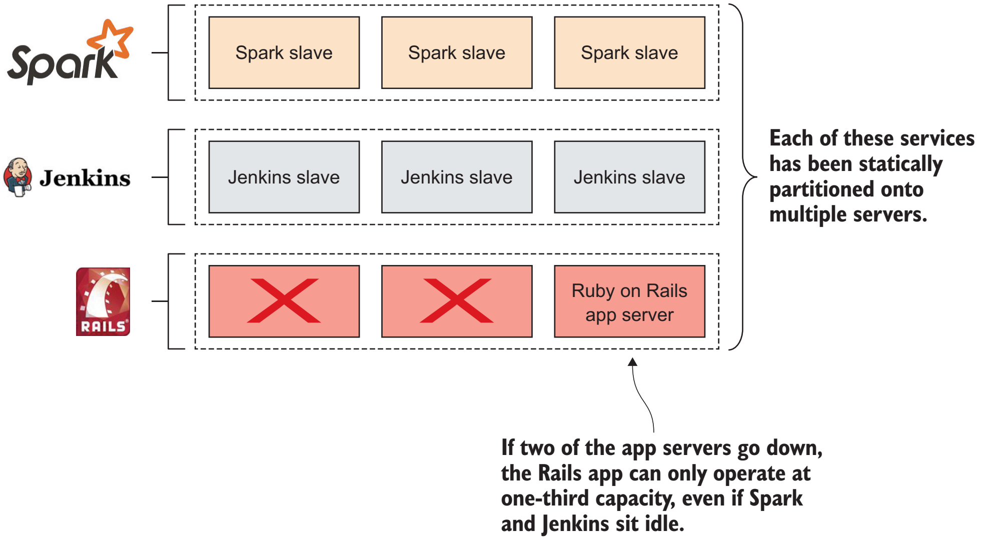

Spark
Spark slave
Spark slave
Spark slave
Jenkins
Jenkins slave
Jenkins slave
Jenkins slave
RAILS
Ruby on Rails app server
Each of these services has been statically partitioned onto multiple servers.
If two of the app servers go down, the Rails app can only operate at one-third capacity, even if Spark and Jenkins sit idle.

Figure 1.5 Three applications statically partitioned in a datacenter

Now consider solving the aforementioned scaling scenario by using Mesos, as shown in figure 1.6. You can see that you'd use these same machines in the datacenter to focus on running applications instead of virtual machines. The applications could run on any machine with available resources. If you need to scale, you add servers to the *Mesos* cluster, instead of adding machines to multiple clusters. If a single Mesos node goes offline, no particular impact occurs to any one service.

These services are run on Mesos, which dynamically schedules them within the cluster based on available capacity.

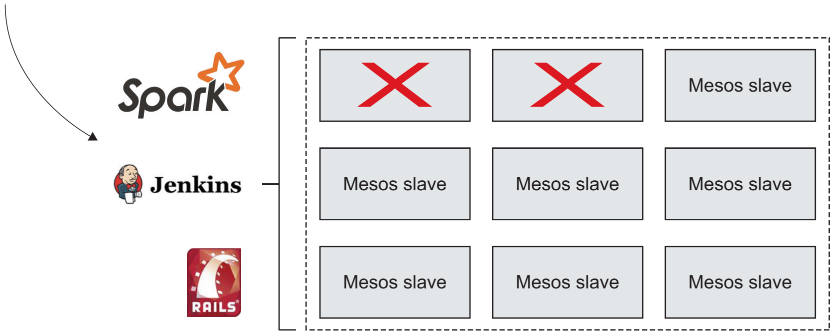


Figure 1.6 Three applications running on a Mesos cluster

Consider these small differences across hundreds or thousands of servers. Instead of trying to guess how many servers you need for each service and provision them into several static clusters, you're able to allow these services to dynamically request the compute, memory, and storage resources they need to run. To continue scaling, you add new machines to your Mesos cluster, and the applications running on the cluster scale to the new infrastructure. Operating a single, large computing cluster in this manner has several advantages:

- You can easily provision additional cluster capacity.
- You can be less concerned about where services are running.
- You can scale from several nodes to thousands.
- The loss of several servers doesn't severely degrade any one service.

1.2.2 Deploying applications

As we discussed previously, one of the major differences—and benefits—of deploying applications on a Mesos cluster is multitenancy. Not unlike a virtualization hypervisor running multiple virtual machines on a physical server, Mesos allows multiple applications to run on a single server in isolated environments, using either Linux *cgroups* or

Docker *containers*. Instead of having multiple environments (one each for development, staging, and production), the entire datacenter becomes a platform on which to deploy applications.

Where Mesos is commonly referred to—and acts as—a distributed *kernel*, other Mesos frameworks help users run long-running and scheduled tasks, similar to the init and Cron systems, respectively. You'll learn more about these frameworks (Marathon, Chronos, and Aurora) and how to deploy applications on them later in this book.

Consider the power of what I've described so far: Mesos provides fault tolerance out of the box. Instead of a systems administrator getting paged when a single server goes offline, the cluster will automatically start the failed job elsewhere. The sysadmin needs to be concerned only if a certain percentage of machines goes offline in the datacenter, as that might signal a larger problem. As such, with the correct placement and redundancy in place, scheduled maintenance can occur at any time.

1.3 The Mesos distributed architecture

To provide services at scale, Mesos provides a distributed, fault-tolerant architecture that enables fine-grained resource scheduling. This architecture comprises three components: *masters*, *slaves*, and the applications (commonly referred to as *frameworks*) that run on them. Mesos relies on Apache ZooKeeper, a distributed database used specifically for coordinating leader election within the cluster, and for leader detection by other Mesos masters, slaves, and frameworks.

In figure 1.7, you can see how each of these architecture components works together to provide a stable platform on which to deploy applications. I'll break it down for you in the sections that follow the diagram.

1.3.1 Masters

One or more Mesos masters are responsible for managing the Mesos slave daemons running on each machine in the cluster. Using ZooKeeper, they coordinate which node will be the *leading master*, and which masters will be on standby, ready to take over if the leading master goes offline.

The leading master is responsible for deciding which resources to offer to a particular framework using a pluggable *allocation module,* or scheduling algorithm, to distribute *resource offers* to the various schedulers. The scheduler can then either accept or reject the offer based on whether it has any work to be performed at that time.

A Mesos cluster requires a minimum of one master, and three or more are recommended for production deployments to ensure that the services are highly available. You can run ZooKeeper on the same machines as the Mesos masters themselves, or use a standalone ZooKeeper cluster. Chapter 3 goes into more detail about the sizing and deploying of Mesos masters.

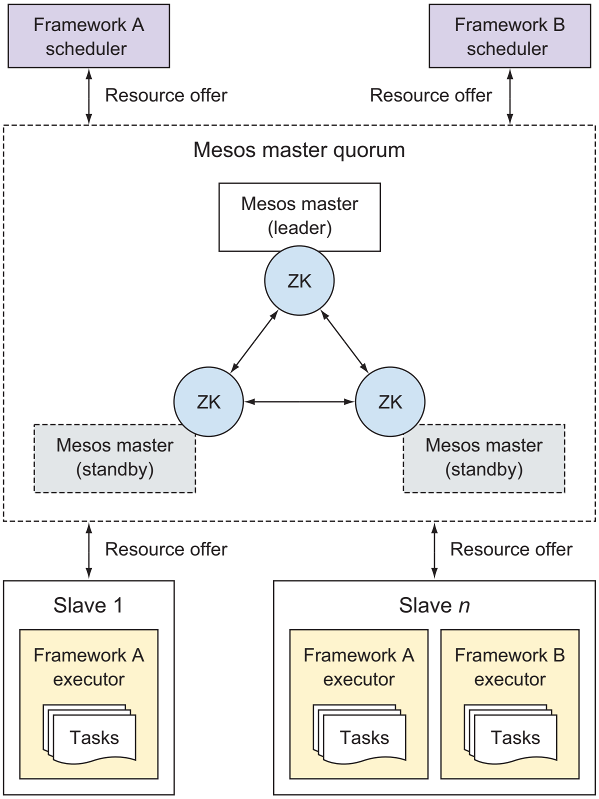


Figure 1.7 The Mesos architecture consists of one or more masters, slaves, and frameworks.

1.3.2 Slaves

The machines in a cluster responsible for executing a framework's tasks are referred to as Mesos *slaves*. They query ZooKeeper to determine the leading Mesos master and advertise their available CPU, memory, and storage resources to the leading master in the form of a resource offer. When a scheduler accepts a resource offer from the Mesos master, it then launches one or more *executors* on the slave, which are responsible for running the framework's tasks.

Mesos slaves can also be configured with certain attributes and resources, which allow them to be customized for a given environment. *Attributes* refer to key/value pairs that might contain information about the node's location in a datacenter, and *resources* allow a particular slave's advertised CPU, memory, and disk to be overridden with user-provided values, instead of Mesos automatically detecting the available resources on the slave. Consider the following example attributes and resources:

```
--attributes='datacenter:pdx1;rack:1-1;os:rhel7'
--resources='cpu:24;mem:24576;disk:409600'
```

I've configured this particular Mesos slave to advertise its datacenter; location within the datacenter; operating system; and user-provided CPU, memory, and disk resources. This information is especially useful when trying to ensure that applications stay online during scheduled maintenance. Using this information, a datacenter operator could take an entire rack (or an entire row!) of machines offline for scheduled maintenance without impacting users. Chapter 4 covers this (and more) in the Mesos slave configuration section.

1.3.3 Frameworks

As you learned earlier, a *framework* is the term given to any Mesos application that's responsible for scheduling and executing tasks on a cluster. A framework is made up of two components: a scheduler and an executor.

> **TIP** A list of frameworks known to exist at the time of writing is included in appendix B.

Scheduler

A *scheduler* is typically a long-running service responsible for connecting to a Mesos master and accepting or rejecting resource offers. Mesos delegates the responsibility of scheduling to the framework, instead of attempting to schedule all the work for a cluster itself. The scheduler can then accept or reject a resource offer based on whether it has any tasks to run at the time of the offer. The scheduler detects the leading master by communicating with the ZooKeeper cluster, and then registers itself to that master accordingly.

Executor

An *executor* is a process launched on a Mesos slave that runs a framework's tasks on a slave. As of this writing, the built-in Mesos executors allow frameworks to execute shell scripts or run Docker containers. New executors can be written using Mesos's various language bindings and bundled with the framework, to be fetched by the Mesos slave when a task requires it.

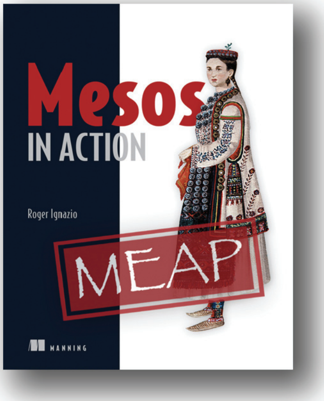
As you've learned, Mesos provides a distributed, highly available architecture. Masters schedule work to be performed on the cluster, and slaves advertise available resources to the schedulers, which in turn execute tasks on the cluster.

1.4 Summary

In this chapter, you've been introduced to the Apache Mesos project, its architecture, and how it attempts to solve scaling problems and make clustering simple. You've also learned how Mesos deployments compare and contrast with the traditional datacenter, and how an application-centric approach can lead to using resources more efficiently. We've discussed when (and when not) to use Mesos for a given workload, and where you can get help and find more information, should you need it. Here are a few things to remember:

- Mesos abstracts CPU, memory, and disk resources away from underlying systems and presents multiple machines as a single entity.
- Mesos slaves advertise their available CPUs, memory, and disk in the form of resource offers.
- A Mesos framework comprises two primary components: a scheduler and an executor.
- Containers are a lightweight method to provide resource isolation to individual processes.

In the next chapter, I'll walk you through a real-world example of how Mesos allows for more efficient resource use, and how you might run applications in your own datacenter by building on projects in the Mesos ecosystem.



The modern "data center" is a complex arena, with physical and virtual servers, multiple OS environments, and complex networking that frequently spans multiple locations. When you throw Docker and container-based systems like CoreOS and Project Atomic into the mix, along with the increasingly important requirement for infrastructure automation, the need to simplify the data center has never been greater. Mesos, an innovative open-source cluster management platform, transforms the whole data center into a single pool of compute, memory and storage resources that you can allocate, automate, and scale as if you're working with a single super-computer. Mesos is an ideal environment for deploying containerized applications at scale, and it's generating huge buzz in the big data world as a saner environment for running Spark and Hadoop.

Mesos in Action introduces readers to the Apache Mesos cluster manager and the concept of application-centric infrastructure. It guides you from your first steps in deploying a highly available Mesos cluster through deploying applications in production and writing native Mesos frameworks. You'll learn how to scale to thousands of nodes, while providing resource isolation between processes using Linux and Docker containers. You'll also learn practical techniques for deploying applications using popular key frameworks, including Marathon, Chronos, and Aurora. Along the way, you'll get a good look into Mesos internals, including fault tolerance, slave attributes, and resource scheduling and Mesos administration, including logging, monitoring, framework authorization, and slave recovery.

What's inside

- Spinning up your first Mesos cluster
- Deploying containerized applications on Mesos
- Scheduling, resource administration, and logging
- Deploy applications using the popular Marathon, Chronos, and Aurora frameworks
- Writing custom Mesos frameworks using Python

Readers need to be familiar with the core ideas of data center administration, including networking, virtualization, and application deployment on Linux systems. The Python-based code examples should be clear to readers using Mesos bindings for other popular languages, including C++, Go, and Scala.

RabbitMQ in Depth

Decoupling is a popular pattern when architecting systems for the cloud. By decoupling different parts of your system, you end up with scalability and high availability. RabbitMQ, a message-oriented middleware supports you when creating distributed software architectures. *RabbitMQ in Depth* gives you valuable insights into decoupling in general and RabbitMQ in particular. The first chapter, “Foundational RabbitMQ,” covers features and benefits from using RabbitMQ for your future cloud application.

Chapter 1 from *RabbitMQ in Depth* by Gavin M. Roy.

Foundational RabbitMQ

This chapter covers:

- Unique features of RabbitMQ
- Why RabbitMQ is becoming a popular choice for the centerpiece of messaging-based architectures
- The basics of the Advanced Messaging Queuing Model, RabbitMQ's foundation

Whether your application is in the cloud or in your own data center, RabbitMQ is a lightweight and extremely powerful tool for creating distributed software architectures that ranges from the very simple to the incredibly complex. In this chapter you will learn how RabbitMQ, as messaging-oriented middleware, allows tremendous flexibility in how you approach and solve problems. You will learn about how some companies are using it and key features that make RabbitMQ one of the most popular message brokers today.

1.1 RabbitMQ's features and benefits

RabbitMQ has many features and benefits, the most important ones outlined below:

- Originally developed in a partnership between LShift, LTD, and Cohesive FT as RabbitMQ Technologies, RabbitMQ is now owned by Pivotal Software Inc. and released under the Mozilla Public License. As an *open-source* project written in Erlang, RabbitMQ enjoys freedom and flexibility while leveraging the strength of Pivotals support of the product. Developers and engineers in the RabbitMQ community contribute enhancements and add-ons while Pivotal offers commercial support and a stable home for ongoing product maturation.
- As a message broker that implements *platform and vendor neutral* AMQP specification, there are clients available for almost any programming language and on all major computer platforms.
- It is *lightweight*, requiring under 40 MB of RAM to run the core RabbitMQ application, along with plugins such as the Management UI. Note that adding messages to queues can and will increase its memory usage.
- *Client libraries target most modern programming languages* on multiple platforms, and RabbitMQ is a compelling broker. There are no vendor or language lock-ins when choosing how to write programs that talk to RabbitMQ. In fact, it is not uncommon to see RabbitMQ used as the centerpiece between applications written in different languages. RabbitMQ provides a useful bridge that allows for languages like Java, Ruby, Python, PHP, JavaScript and C# to share data across operating systems and environments.
- RabbitMQ provides *flexibility in controlling the trade-offs of reliable messaging* with message throughput and performance. Because it's not a "one size fits all" type of application, messages can designate if they should be persisted to disk prior to delivery. If setup in a cluster, queues can be highly-available, spanning multiple servers to ensure that messages are not lost in case of server failure.
- Because not all network topologies and architectures are the same, RabbitMQ provides for messaging in low-latency environments and *plugins for higher-latency environments* such as the Internet. This allows for RabbitMQ to be clustered on the same local network and share federated messages across multiple data-centers.
- As a center point for application integrations, RabbitMQ provides a flexible plugin system. As an example, there are *third-party plugins* for storing messages directly into databases, using RabbitMQ directly for database writes.
- In RabbitMQ, *security* is provided in multiple layers. Client connections can be secured by enforcing SSL-only communication and client certificate validation. User access can be managed at the virtual-host level, providing isolation of messages and resources at a high-level. In addition, access to configuration capabilities, the reading from queues and writing to exchanges, is managed by regular expression (regex) pattern matching. Finally, plugins can be used for integration into external authentication systems like LDAP.

While we will explore the features on this list in later chapters, I would like to focus on the two most foundational features of RabbitMQ: the language it is programmed in (Erlang), and the model it is based on (the Advanced Message Queuing Model), a specification that defines much of the RabbitMQ lexicon and behavior.

1.1.1 RabbitMQ and Erlang

As a highly performant, stable, and message broker that clusters well, it is no surprise that RabbitMQ has found a home in such mission-critical environments as the centerpiece of large scale messaging architectures. It was written in Erlang, the telco-grade, functional programming language designed at the Ericsson Computer Science Laboratory in the mid-to-late 1980's. Erlang was designed to be a distributed, fault-tolerant, soft real-time system for applications that require 99.999% uptime. As a language and runtime system, Erlang focuses on lightweight processes that pass messages amongst each other providing a high level of concurrency with no shared state.

> **REAL-TIME SYSTEM** A real-time system is a hardware platform, software platform, or a combination of both that has requirements defined for when it must return a response from an event. A soft real-time system will sacrifice less important deadlines for executing tasks in favor of more important ones.

Erlang's design around concurrent processing and message passing makes it a natural choice for a message broker like RabbitMQ: As an application, a message broker maintains concurrent connections, routes messages, and manages their state. In addition, Erlang's distributed communication architecture makes it a natural for RabbitMQ's clustering mechanism. Servers in a RabbitMQ cluster make use of Erlang's *inter-process communication* (IPC) system, offloading the functionality that many competing message brokers have to implement to add clustering capabilities (figure 1.1).

Despite the advantages RabbitMQ has using Erlang, the Erlang environment can be a stumbling block. If this is your first foray into Erlang, check out Appendix B: Just Enough Erlang. In the appendix, you will learn enough to be confident in managing RabbitMQ's configuration files and you will learn how to use Erlang to gather information about RabbitMQ's current runtime state.

1.1.2 RabbitMQ and AMQP

When RabbitMQ was originally released in 2007, interoperability, performance, and stability were the primary goals in mind during development, and RabbitMQ was one of the first message brokers to implement the Advanced Message Queuing Protocol (AMQP) specification. By all appearances, it set out to be the reference implementation. Split into two parts, the AMQP specification defines not only the wire protocol for talking to RabbitMQ, but the logical model that outlines RabbitMQ's core functionality.

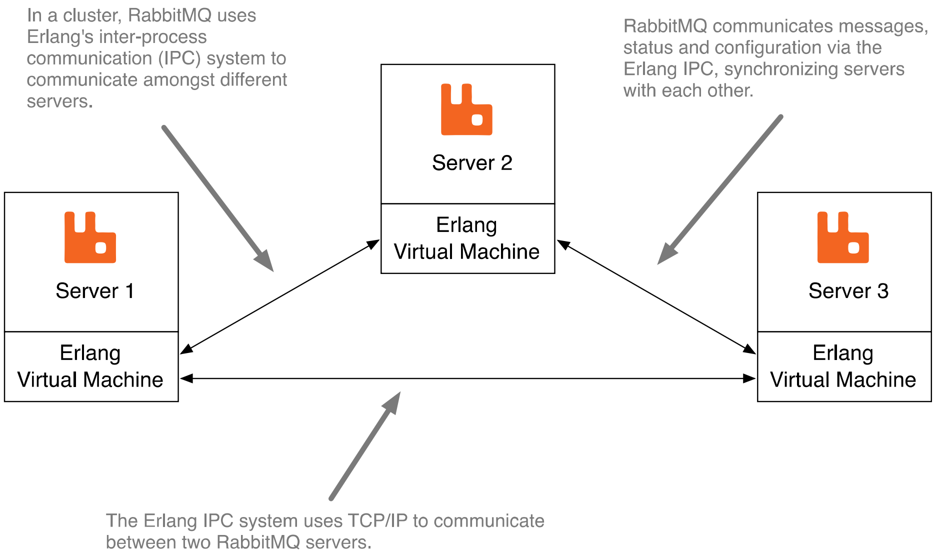

In a cluster, RabbitMQ uses Erlang's inter-process communication (IPC) system to communicate amongst different servers.
Server 2
Erlang Virtual Machine
RabbitMQ communicates messages, status and configuration via the Erlang IPC, synchronizing servers with each other.
Server 1
Erlang Virtual Machine
Server 3
Erlang Virtual Machine
The Erlang IPC system uses TCP/IP to communicate between two RabbitMQ servers.

Figure 1.1 RabbitMQ clusters use the native Erlang inter-process communication mechanism in the Virtual Memory (VM) for cross-node communication, sharing state information and allowing for messages to be published and consumed across the entire cluster.

> **NOTE** There are multiple versions of the AMQP specification. For the purposes this book, we will focus only on AMQP 0-9-1. While newer versions of RabbitMQ support AMQP 1.0 as a plugin extension, the core RabbitMQ architecture is more closely related to AMQP 0-8 and 0-9-1. The AMQP specification is primarily comprised of two documents, a top-level document that describes both the AMQ model and the AMQ protocol and a more detailed document that provides varying levels of information about every class, method, property and field. More information about AMQP, including the specification documents may be found at http://www.amqp.org.

There are multiple popular message brokers and messaging protocols, and it is important that you consider the impact the protocol and broker will have on your application. While RabbitMQ supports AMQP, it also supports other protocols, such as MQTT, Stomp, and XMPP. RabbitMQ's protocol neutrality and plugin extensibility make it a good choice for multi-protocol application architectures when compared to other popular message brokers.

It is RabbitMQ's roots in the AMQP specification that outline its primary architecture and communication methodologies. This is an important distinction when evaluating RabbitMQ against other message brokers. As with AMQP, RabbitMQ set out to be a vendor-neutral, platform-independent solution for the complex needs that messaging oriented architectures demand, such as flexible message routing, configurable message durability, and inter-datacenter communication, to name a few.

1.2 Who is using RabbitMQ, and how?

As an open-source software package, RabbitMQ is rapidly gaining mainstream adoption and powers some of the largest, most trafficked websites on the Internet. Today, RabbitMQ is known to run in many different environments and at many different types of companies and organizations:

- Reddit, the popular online community, utilizes RabbitMQ heavily in the core of their application platform, which serves billions of web pages per month. When a user registers on the site, submits a news post, or votes on a link, a message is published into RabbitMQ for asynchronous processing by consumer applications.
- NASA chose RabbitMQ to be message-broker for Nebula, a centralized server management platform for their server infrastructure that grew into the OpenStack platform, a very popular software platform for building private and public cloud services.
- RabbitMQ sits at the core of Agoura Games' community-oriented online gaming platform and routes large volumes of real-time single and multiplayer game data and events.
- For the Ocean Observations Initiative, RabbitMQ routes mission-critical physical, chemical, geological, and biological data to a distributed network of research computers. The data, collected from sensors in the Southern, Pacific, and Atlantic oceans, are integral to a National Science Foundation project that involves building a large-scale network of sensors in the ocean and seafloor.
- Rapportive, a GMail add-on that places detailed contact information inside the inbox, uses RabbitMQ as the glue for its data processing systems. Billions of messages pass through RabbitMQ monthly to provide data to Rapportive's web-crawling engine and analytics system, and to offload long-running operations from its web servers.
- MercadoLibre, the largest e-commerce ecosystem in Latin America, use RabbitMQ at the heart of their Enterprise Service Bus (ESB) architecture, decoupling their data from tightly coupled applications. This allows for flexible integrations with various components in their application architecture.
- Google's *AdMob* mobile advertising network used RabbitMQ at the core of their RockSteady project to do real-time metrics analysis and fault-detection by funneling a fire hose of messages through RabbitMQ into Esper, the complex-event-processing system.
- India's biometric database system, *Aandhaar,* leverages RabbitMQ to process data at various stages in its workflow, delivering data to their monitoring tools, data warehouse, and their Hadoop based data processing system. *Aandhaar* is a system for providing an online portable identity system for every single resident of India, covering 1.2 billion people.

As you can see, RabbitMQ is not only used by some of the largest sites on the Internet, it has found its way into academia for large scale scientific research, and NASA found it fitting to use RabbitMQ at the core of their network infrastructure management stack. As these examples show, RabbitMQ has been used in mission-critical applications in many different environments and industries with tremendous success.

1.3 The advantages of loosely coupled architectures

When I first started to implement a messaging based architecture, I was looking for a way to decouple needed database updates when a member logged into a website. The website had grown very quickly and was not initially designed to scale well. As a user logged into the website, several database servers needed to be updated with a timestamp when the member logged in (figure 1.2). This timestamp needed to be updated in real-time, as the most engaging activities on the site were driven in part by the timestamp value. Upon login, members were given preferential status in social games to those users who were actively online at any given time.

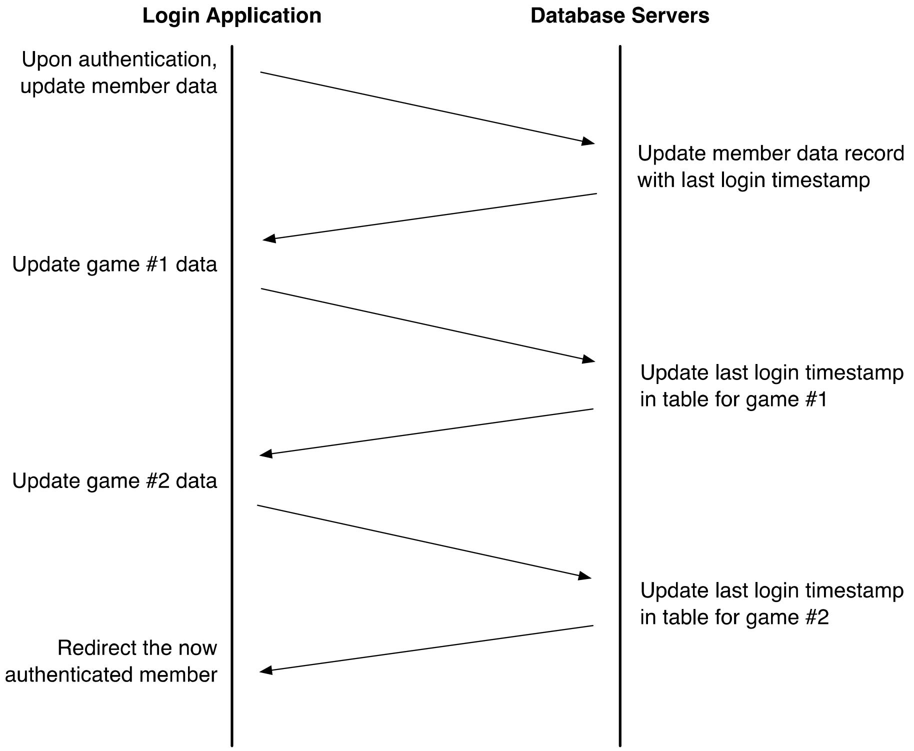

Login Application
Database Servers
Upon authentication, update member data
Update member data record with last login timestamp
Update game #1 data
Update last login timestamp in table for game #1
Update game #2 data
Update last login timestamp in table for game #2
Redirect the now authenticated member

Figure 1.2 Before: Once a user has logged in, each database is updated with a timestamp sequentially and dependently. The more tables you add increases the time this takes.

As the site continued to grow, the amount of time it took for a member to login also grew. The reason for this was fairly straightforward; when adding a new application that used the member's last login timestamp, its database tables would carry the value to make it as fast as possible by removing cross database joins. To keep the data up to date and accurate, the new data tables would also be updated when the member logged in. It was not long before there were many tables being maintained this way. The performance degradation began to creep up as the database updates were performed serially. Each query updating the member's last login timestamp needed to finish before the next began. Ten queries that were considered performant, each finishing within 50ms, would add up to half a second in database updates alone. All of these queries would have to finish prior to sending the authorization response and redirect back to the user. In addition, any operational issues on a database server compounded the problem. If one database server in the chain of servers started responding slowly or became unresponsive, members could no longer login into the site.

To decouple the user-facing login application from directly writing to the database. I published messages to Message-oriented-middleware or a centralized message broker, which would distribute the message to consumer applications that handle the required database writes. While I first experimented with several different message brokers, ultimately I landed on RabbitMQ as my broker of choice.

> **DEFINITION** Message-oriented-middleware (MOM) is defined as software or hardware infrastructure that allows for the sending and receiving of messages from distributed systems. RabbitMQ fills this role with functionality that provides advanced routing and message distribution, even with wide-area network (WAN) tolerances to support reliable, distributed systems that interconnect with other systems easily.

After decoupling the login process from the required database updates, a new level of freedom was discovered. Members were able to quickly login because we were no longer updating the database as part of the authentication process. Instead a member login message was published with all of the information needed to update any database, and consumer applications were written that updated each database table independently (figure 1.3). This login message would not contain authentication for the member, but instead, only the information needed to maintain the member's last-login status in our databases and applications. This allowed us to horizontally scale database writes with more control. By controlling the number of consumer applications writing for a specific database server, we were able to throttle database writes for servers that had started to strain under the load created by new site growth, while we worked through their own unique scaling issues.

As I detail the advantages of a messaging based architecture, it is important to note that they **(they who?)** could also impact the performance of systems like the login architecture described. Any number of problems may impact publisher performance, from networking issues to RabbitMQ throttling message publishers. When such events happen, your application will see degraded performance. In addition to the horizon-

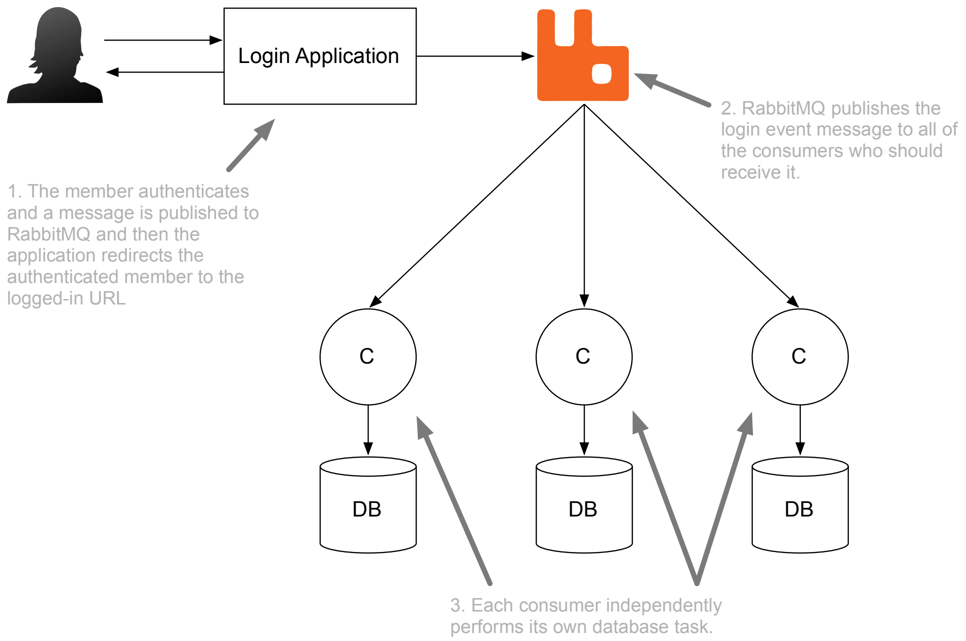

Login Application
2. RabbitMQ publishes the login event message to all of the consumers who should receive it.
1. The member authenticates and a message is published to RabbitMQ and then the application redirects the authenticated member to the logged-in URL
C
C
C
DB
DB
DB
3. Each consumer independently performs its own database task.

Figure 1.3 After: Using RabbitMQ, loosely coupled data is published to each database asynchronously and independently, allowing the login application to proceed without waiting on any database writes.

tal scaling of consumers, it is wise to plan for horizontal scaling of message brokers to allow for better message throughput and publisher performance.

1.3.1 Decoupling your application

The use of messaging-oriented-middleware can provide tremendous advantages for organizations looking to create flexible application architectures that are data centric. By moving to a loosely coupled design using RabbitMQ, application architectures are no longer bound to database write performance and can easily add new applications to act upon the data without touching any of the core applications. Consider figure 1.4 demonstrating the design of a tightly coupled application communicating with a database:

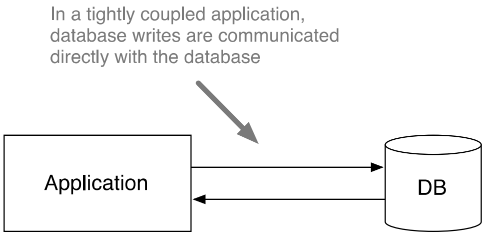

In a tightly coupled application, database writes are communicated directly with the database
Application
DB

Figure 1.4 When communicating with a database, a tightly coupled application must wait for the database server to respond to continue processing.

1.3.2 Decoupling database writes

In a tightly coupled architecture, the application must wait for the database server to respond before it can finish a transaction. This design has the potential to create performance bottlenecks in both synchronous and asynchronous applications. If the database server slows down due to poor tuning or hardware issues, the application will also become slower. Should the database no longer respond, or it crashes, the application will potentially crash as well. By decoupling the database from the application, a loosely coupled architecture is created. In this architecture, RabbitMQ as messaging-oriented-middleware acts as an intermediary for the data prior to some action being taken with it in the database. A consumer application picks up the data from the RabbitMQ server, performing the action with the database (figure 1.5).

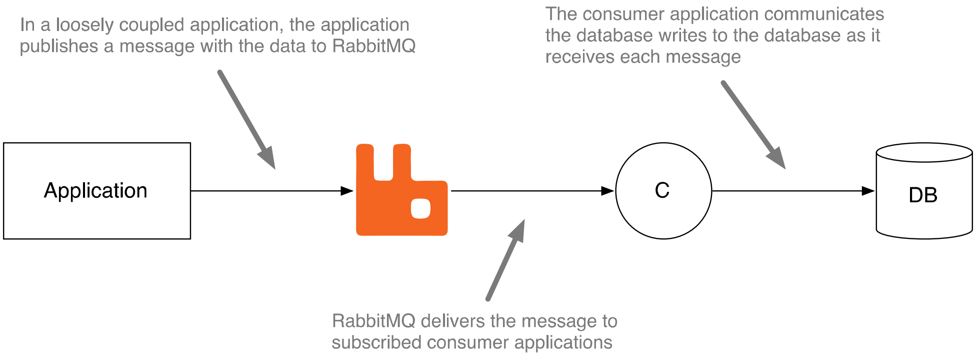

In a loosely coupled application, the application publishes a message with the data to RabbitMQ
The consumer application communicates the database writes to the database as it receives each message
Application
C
DB
RabbitMQ delivers the message to subscribed consumer applications

Figure 1.5 A loosely coupled application allows the application that would have saved the data directly in the database to publish the data to RabbitMQ, allowing for the asynchronous processing of data.

In this model, should a database need to be taken offline for maintenance, or should the write workload become too heavy, you can throttle the consumer application or stop it. Until the consumer is able to receive the message, the data will persist in the queue. The ability to pause or throttle consumer application behavior is just one advantage of using this type of architecture.

1.3.3 Seamlessly adding new functionality

Loosely coupled architectures leveraging RabbitMQ allows data to be repurposed as well. The data written to a database can also be used for other purposes. RabbitMQ will handle all of the duplication of the message content and can route it to multiple consumers for multiple purposes (figure 1.6).

1.3.4 Replication of data and events

Expanding upon this model, RabbitMQ provides built-in tools for cross-datacenter distribution of data, allowing for federated delivery and synchronization of applica-

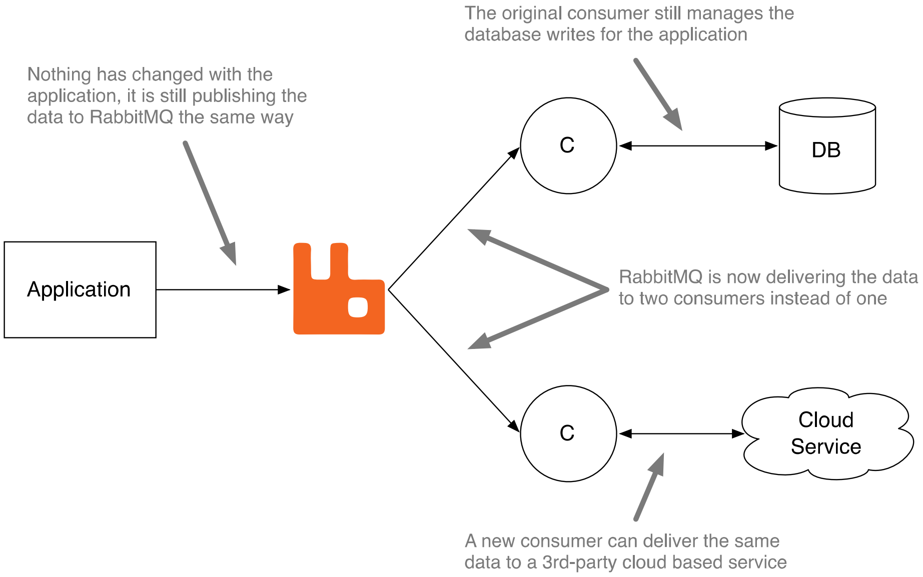

The original consumer still manages the database writes for the application
Nothing has changed with the application, it is still publishing the data to RabbitMQ the same way
C
DB
Application
RabbitMQ is now delivering the data to two consumers instead of one
C
Cloud Service
A new consumer can deliver the same data to a 3rd-party cloud based service

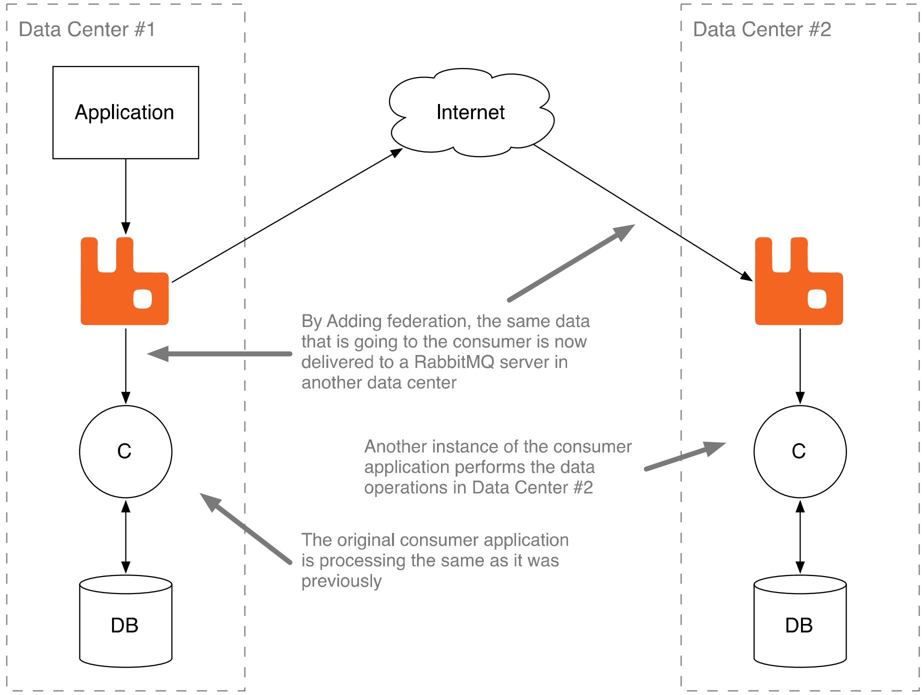
Figure 1.6 By using RabbitMQ, the publishing application does not need to be changed in order to deliver the same data to both a new cloud-based service and the original database.

tions. Federation provides a mechanism that allows RabbitMQ to push messages to remote RabbitMQ instances, accounting for WAN tolerances and network splits. Using the RabbitMQ federation plugin, it is easy to add a RabbitMQ server or cluster to a second data center. This is illustrated in figure 1.7 where the data from the original application can now be processed in two different locations over the Internet.

1.3.5 Multi-Master federation of data and events

Expanding upon this concept, by adding the same front-end application to Data Center #2 and setting the RabbitMQ servers to bi-directionally federate data, you can have highly available applications in different physical locations. Messages from the application in either data center are sent to consumers in both data centers, allowing for redundancy in data storage and processing (figure 1.8). This approach to application architecture can provide a scale-out approach to applications, providing geographic proximity for users and a cost-effective way to distribute your application infrastructure.

NOTE: As with any architecture decision, using messaging-oriented-middleware introduces a degree of operational complexity. Because a message broker becomes a center point in your application design, a new point of failure is introduced. There are strategies, which we will cover in this book, to create highly available solutions to minimize this risk. In addition, adding a message broker creates a new application to manage. Configuration, server resources, and monitoring must be taken into account when weighing the tradeoffs of introducing a message broker to your architecture; I will teach you how to account for these and other concerns as you proceed through the book.

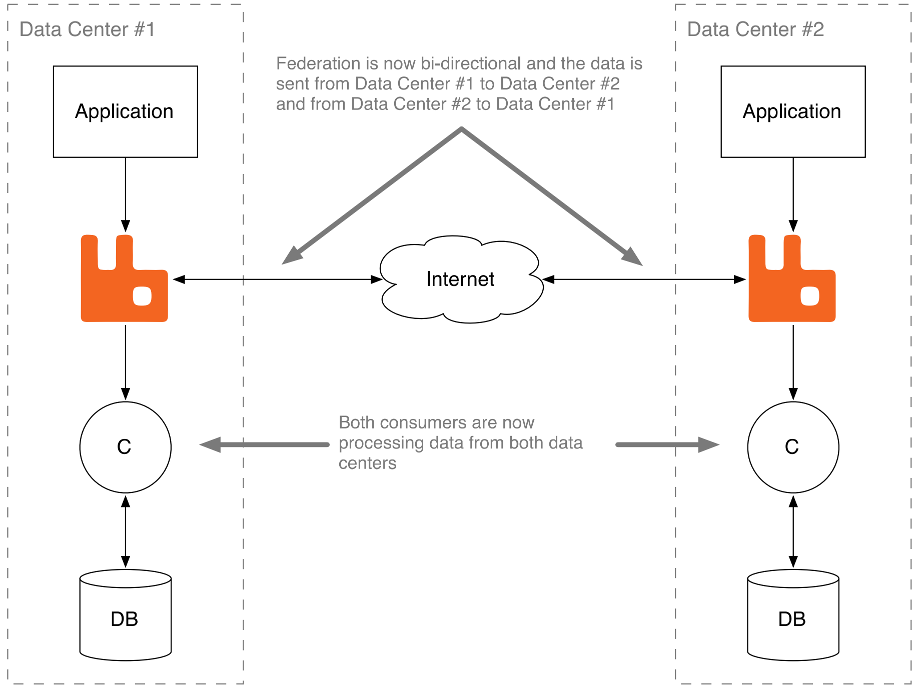


By Adding federation, the same data that is going to the consumer is now delivered to a RabbitMQ server in another data center

Another instance of the consumer application performs the data operations in Data Center #2

The original consumer application is processing the same as it was previously

Figure 1.7 By leveraging RabbitMQ's federation plugin, messages can be duplicated to perform the same work in multiple data centers.



Federation is now bi-directional and the data is sent from Data Center #1 to Data Center #2 and from Data Center #2 to Data Center #1

Both consumers are now processing data from both data centers

Figure 1.8 Bi-directional federation of data allows for the same data events to be received processed in both data centers.

1.4 The Advanced Message Queuing Model

Many of RabbitMQ's strengths and flexibility come from the AMQP specification. Unlike protocols like HTTP and SMTP, the AMQP specification defines not only a network protocol, but it also defines server-side services and behaviors. To have a common way to refer to this information is the Advanced Message Queuing (AMQ) Model. The AMQ model logically defines three abstract components in broker software that determine the routing behavior of messages:

- An *exchange*, the component of the message broker that routes messages to queues
- A *queue*, a data structure on disk or in memory that stores messages
- A *binding*, which tells the exchange which queue messages should be stored in

The flexibility of RabbitMQ comes from the dynamic nature of how messages can be routed through exchanges to queues. These bindings between exchanges and queues, and the message routing dynamics they create, are a foundational component of implementing messaging based architecture. Creating the right structure by using these basic tools in RabbitMQ allows your applications to scale and easily change with the underlying business needs.

EXCHANGES

The first piece of information that RabbitMQ needs in order to route messages to their proper destination is an exchange to route them through. Exchanges are one of three components defined by the AMQ model. An exchange receives messages sent into RabbitMQ and determines where to send them. Exchanges define the routing behaviors that are applied to messages, usually by examining data attributes passed along, or that are contained within the message's properties. RabbitMQ has multiple exchange types, each with different routing behaviors. In addition, it offers a plug-in based architecture for custom exchanges. Figure 1.9 shows a logical view of a publisher sending a message to RabbitMQ, routing a message through an exchange, the first component of the AMQ model.

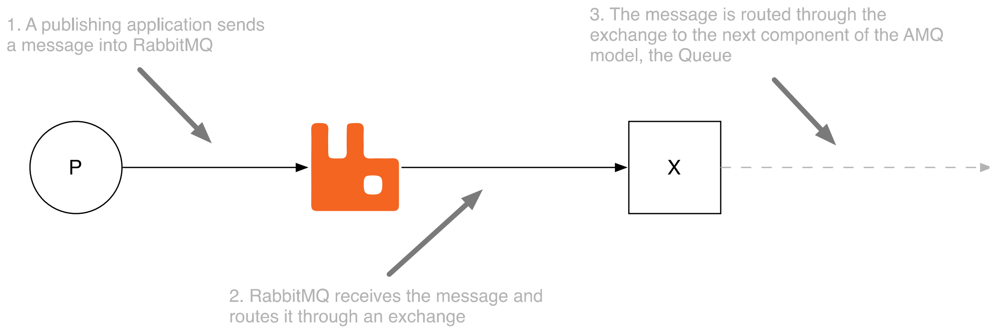

1. A publishing application sends a message into RabbitMQ
3. The message is routed through the exchange to the next component of the AMQ model, the Queue
P
X
2. RabbitMQ receives the message and routes it through an exchange

Figure 1.9 When a publisher sends a message into RabbitMQ, it first goes to an exchange.

Queues

A queue is responsible for storing received messages and may contain configuration information that defines what it is able to do with a message. Queues may hold messages in RAM only or it may persist them to disk prior to delivering messages from a queue in first-in, first-out (FIFO) order.

Bindings

To define a relationship between queues and exchanges, the AMQ model defines a *binding*. In RabbitMQ, bindings or *binding-keys* tell an exchange which queues to deliver messages to. For some exchange types it will also instruct the exchange to filter which messages it can deliver to a queue. When publishing a message to an exchange, applications use a *routing*-key attribute. Sometimes this may be a queue name, at other times it may be a string that semantically describes the message. When a message is evaluated by an exchange to be routed to the appropriate queues, the message's routing-key is evaluated against the binding-key (figure 1.10). In other words, the binding-key is the glue that binds a queue to an exchange and the routing-key is the criteria that is evaluated against it.

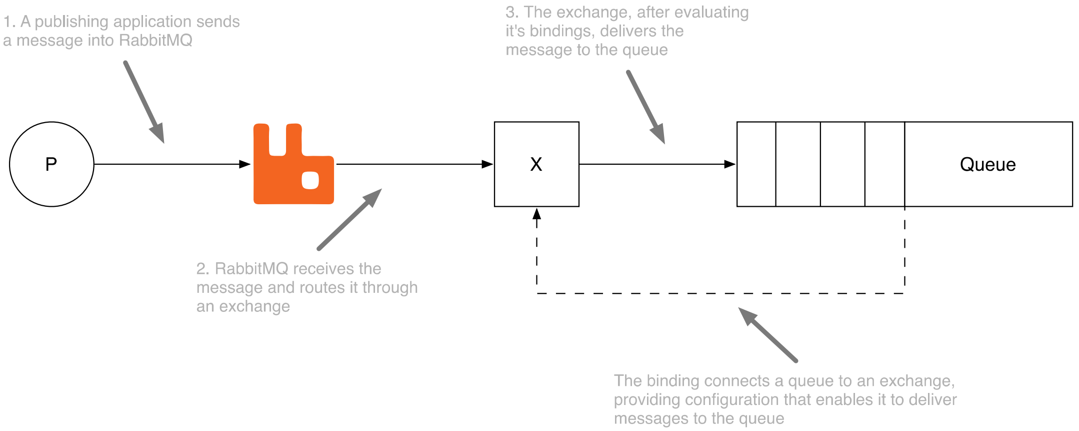

1. A publishing application sends a message into RabbitMQ
3. The exchange, after evaluating it's bindings, delivers the message to the queue
P
X
Queue
2. RabbitMQ receives the message and routes it through an exchange
The binding connects a queue to an exchange, providing configuration that enables it to deliver messages to the queue

Figure 1.10 A queue is bound to an exchange, providing the information the exchange needs to route a message to it.

In the simplest of scenarios, the routing key may be the queue name, though this varies with each exchange type. In RabbitMQ, each exchange type is likely to treat routing-keys in a different way; some exchanges invoke simple equality checks and others use more complex pattern extractions from the routing-key. There is even an exchange type that ignores the routing-key outright in favor of other information in the message properties.

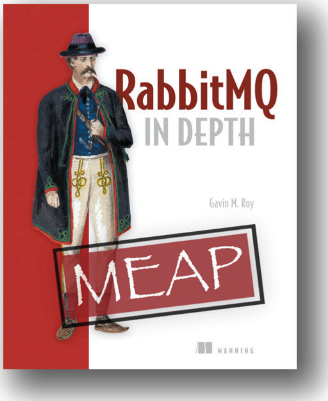
In addition to binding queues to exchanges, as defined in the AMQ model, RabbitMQ extends the AMQP specification to allow exchanges to bind to other exchanges. This feature creates a great deal of flexibility in creating different routing patterns for messages. You will learn more about routing patterns available when you use exchanges, and about exchange-to-exchange bindings in chapter 6, Common Messaging Patterns.

1.5 Summary

RabbitMQ, as messaging-oriented-middleware, is an exciting technology that enables operational flexibility that is otherwise difficult to achieve without the loosely coupled application architecture it enables. By diving deep into RabbitMQ's AMQP foundation and behaviors, this book is a valuable reference, providing insight into how your applications can leverage its robust and powerful features. In particular, you will soon learn how to publish messages, and use the dynamic routing features in RabbitMQ to selectively sip from the fire hose of data your application can send; data that once may have been deeply buried in tightly coupled code and processes in your environment.

Whether you are an application developer or a high-level application architect, it is advantageous to have deep level knowledge about how your applications can benefit from RabbitMQ's diverse functionality. Thus far, you have already learned the most foundational concepts that comprise the Advanced Message Queuing Model. Expanding on these concepts in Part 1 of this book, you will learn about the Advanced Message Queuing Protocol and how it defines the core of RabbitMQ's behavior.

Because this book will be hands-on with the goal of imparting the knowledge required to use RabbitMQ, the most demanding of environments, you will start working with code in the next chapter. By learning "how to speak Rabbit" you will leverage the fundamentals of the Advanced Message Queuing Protocol, writing code to send and receive messages with RabbitMQ. To speak Rabbit, you will be using a Python based library called *rabbitpy,* a library that was written specifically for the code examples in this book; I'll introduce it to you in the next chapter. Even if you are an experienced developer who has written applications that communicate with RabbitMQ, you should at least browse through the next chapter to understand what is happening at the protocol level when you are using RabbitMQ via the AMQP protocol.



Any large application needs an efficient way to handle the constant messages passing between components in the system. Billed as "messaging that just works," the RabbitMQ message broker initially appeals to developers because it's lightweight, easy to set up, and low maintenance. They stick with it, though, because it's powerful, fast, and up to nearly anything you can throw at it. This book takes you beyond the basics and explores the challenges of clustering and distributing messages across enterprise-level data-centers using RabbitMQ.

RabbitMQ in Depth is a practical guide to building and maintaining message-based systems. This book covers detailed architectural and operational use of RabbitMQ with an emphasis on not just how it works but why it works the way it does. You'll find examples and detailed explanations of everything from low-level communication to integration with third-party systems. You'll also find the insights you need to make core architectural choices and develop procedures for effective operational management.

What's inside

- Understanding the AMQP model
- Communicating via MQTT, Stomp, and HTTP
- Valuable troubleshooting techniques
- Integrating with Java technologies like Hadoop and Esper
- Database integrations with PostgreSQL and Riak

Written for programmers with a basic understanding of messaging oriented systems and RabbitMQ.

Netty in Action

Usually cloud computing is heavily reliant on networking. Distributed systems offer scalability and high-availability by distributing workloads on server fleets. Communication between different applications and other systems becomes more and more important. But developing applications that are relying heavily on networking is. The Netty framework makes it easier to create high-performance networking applications, so we've selected this chapter from *Netty in Action*, which should give you an idea where Netty could fit in your own projects.

Chapter 14 from *Netty in Action* by Norman Maurer and Marvin Allen Wolfthal.

Case studies, part 1

This chapter covers

- Droplr
- Firebase
- Urban Airship

In this chapter we'll present the first of two sets of case studies contributed by companies that have used Netty extensively in their internal infrastructure. We hope that these examples of how others have utilized the framework to solve real-world problems will broaden your understanding of what you can accomplish with Netty.

NOTE The author or authors of each study were directly involved in the project they discuss.

14.1 Droplr—building mobile services

Bruno de Carvalho, Lead Architect

At Droplr we use Netty at the heart of our infrastructure, in everything from our API servers to auxiliary services.

This is a case study on how we moved from a monolithic and sluggish LAMP[1] application to a modern, high-performance and horizontally distributed infrastructure, implemented atop Netty.

14.1.1 How it all started

When I joined the team, we were running a LAMP application that served both as the front end for users and as an API for the client applications—among which, my reverse-engineered, third-party Windows client, *windroplr*.

Windroplr went on to become *Droplr for Windows*, and I, being mostly an infrastructure guy, eventually got a new challenge: completely rethink Droplr's infrastructure.

By then Droplr had established itself as a working concept, so the goals were pretty standard for a 2.0 version:

- Break the monolithic stack into multiple horizontally scalable components
- Add redundancy to avoid downtime
- Create a clean API for clients
- Make it all run on HTTPS

Josh and Levi, the founders, asked me to "make it fast, whatever it takes."

I knew those words meant more than making it *slightly faster* or even *a lot faster*. "Whatever it takes" meant *a full order of magnitude faster*. And I knew then that Netty would eventually play an important role in this endeavor.

14.1.2 How Droplr works

Droplr has an extremely simple workflow: drag a file to the app's menu bar icon and Droplr uploads the file. When the upload completes, Droplr copies a short URL to the file—the *drop*—to the clipboard.

That's it. Frictionless, instant sharing.

Behind the scenes, *drop* metadata is stored in a database—creation date, name, number of downloads, and so on—and the files are stored on Amazon S3.

14.1.3 Creating a faster upload experience

The upload flow for Droplr's first version was woefully naïve:

1. Receive upload
2. Upload to S3
3. Create thumbnails if it's an image
4. Reply to client applications

A closer look at this flow quickly reveals two choke points on steps 2 and 3. No matter how fast the upload from the client to our servers, the creation of a drop would always go through an annoying hiatus after the actual upload completed, until the successful

[1] An acronym for a typical application technology stack; originally Linux, Apache Web Server, MySQL, and PHP.

response was received—because the file would still need to be uploaded to S3 and have its thumbnails generated.

The larger the file, the longer the hiatus. For very large files the connection would eventually time out waiting for the okay from the server. Back then Droplr could offer uploads of only up to 32 MB per file because of this very problem.

There were two distinct approaches to cut down upload times:

- Approach A, optimistic and apparently simpler (figure 14.1):
  - Fully receive the file
  - Save to the local filesystem and immediately return success to client
  - Schedule an upload to S3 some time in the future

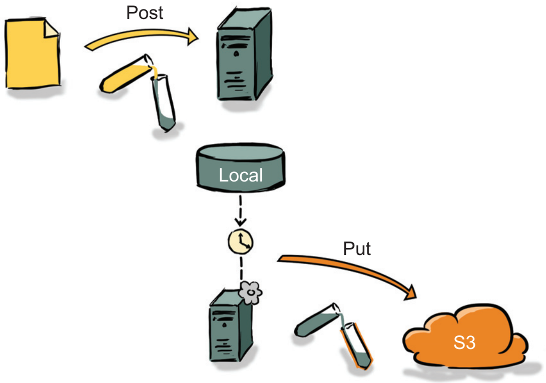


Figure 14.1 Approach A, optimistic and apparently simpler

- Approach B, safe but complex (figure 14.2):
  - Pipe the upload from the client directly to S3, in real time (streaming)

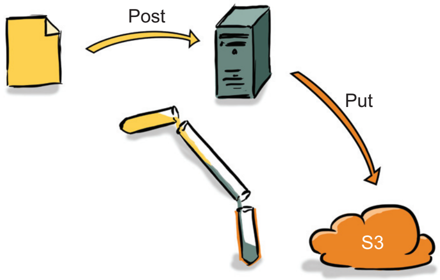


Figure 14.2 Approach B, safe but complex

The optimistic and apparently simpler approach

Returning a short URL after receiving the file creates an expectation—one could even go as far as calling it *an implicit contract*—that the file is immediately available at that URL. But there is no guarantee that the second stage of the upload (actually pushing the file to S3) will ultimately succeed, and the user could end up with a broken link that might get posted on Twitter or sent to an important client. This is unacceptable, even if it happens on one in every hundred thousand uploads.

Our current numbers show that we have an upload failure rate slightly below 0.01% (1 in every 10,000), the vast majority being connection timeouts between client and server before the upload actually completes.

We could try to work around it by serving the file from the machine that received it until it is finally pushed to S3, but this approach is in itself a can of worms:

- If the machine fails before a batch of files is completely uploaded to S3, the files would be forever lost.
- There would be synchronization issues across the cluster ("Where is the file for this drop?").
- Extra, complex logic would be required to deal with edge cases, and this keeps creating more edge cases.

Thinking through all the pitfalls with every workaround, I quickly realized that it's a classic hydra problem—for each head you chop off, two more appear in its place.

THE SAFE BUT COMPLEX APPROACH

The other option required low-level control over the whole process. In essence, we had to be able to

- Open a connection to S3 while receiving the upload from the client.
- Pipe data from the client connection to the S3 connection.
- Buffer and throttle both connections:
  - Buffering is required to keep a steady flow between both client-to-server and server-to-S3 legs of the upload.
  - Throttling is required to prevent explosive memory consumption in case the server-to-S3 leg of the upload becomes slower than the client-to-server leg.
- Cleanly roll everything back on both ends if things went wrong.

It seems conceptually simple, but it's hardly something your average webserver can offer. Especially when you consider that in order to throttle a TCP connection, you need low-level access to its socket.

It also introduced a new challenge that would ultimately end up shaping our final architecture: deferred thumbnail creation.

This meant that whichever technology stack the platform ended up being built upon, it had to offer not only a few basic things like incredible performance and stability but also the flexibility to go *bare metal* (read: *down to the bytes*) if required.

14.1.4 The technology stack

When kick-starting a new project for a webserver, you'll end up asking yourself, "Okay, so what frameworks are the cool kids using these days?" I did too.

Going with Netty wasn't a *no-brainer*; I explored plenty of frameworks, having in mind three factors that I considered to be paramount:

- *It had to be fast.* I wasn't about to replace a low-performance stack with another low-performance stack.
- *It had to scale.* Whether it had 1 or 10,000 connections, each server instance would have to be able to sustain throughput without crashing or leaking memory over time.
- *It had to offer low-level data control.* Byte-level reads, TCP congestion control, the works.

Factors 1 and 2 pretty much excluded any noncompiled language. I'm a sucker for Ruby and love lightweight frameworks like Sinatra and Padrino, but I knew the kind of performance I was looking for couldn't be achieved by building on these blocks.

Factor 2, on its own, meant that whatever the solution, it couldn't rely on blocking I/O. By this point in the book, you certainly understand why non-blocking I/O was the only option.

Factor 3 was trickier. It meant finding the perfect balance between a framework that would offer low-level control of the data it received, but at the same time would be fast to develop with and build upon. This is where language, documentation, community, and other success stories come into play.

At this point I had a strong feeling Netty was my weapon of choice.

THE BASICS: A SERVER AND A PIPELINE

The server is merely a `ServerBootstrap` built with an `NioServerSocketChannelFactory`, configured with a few common handlers and an HTTP `RequestController` at the end, as shown here.

Listing 14.1 Setting up the `ChannelPipeline`

```
pipelineFactory = new ChannelPipelineFactory() {
    @Override
    public ChannelPipeline getPipeline() throws Exception {
        ChannelPipeline pipeline = Channels.pipeline();
        pipeline.addLast("idleStateHandler", new IdleStateHandler(...));
        pipeline.addLast("httpServerCodec", new HttpServerCodec());
        pipeline.addLast("requestController",
            new RequestController(...));
        return pipeline;
    }
};
```

IdleStateHandler shuts down inactive connections

HttpServerCodec converts incoming bytes to HttpRequests and outgoing HttpResponses to bytes

Adds a RequestController to the pipeline

The `RequestController` is the only custom Droplr code in the pipeline and is probably the most complex part of the whole webserver. Its role is to handle initial request validations and, if all is well, route the request to the appropriate request handler. A new instance is created for every established client connection and lives for as long as that connection remains active.

The request controller is responsible for

- Handling load peaks
- HTTP pipeline management
- Setting up a context for request handling
- Spawning new request handlers
- Feeding request handlers
- Handling internal and external errors

Here is a quick rundown of the relevant parts of the `RequestController`.

Listing 14.2 The `RequestController`

```
public class RequestController
    extends IdleStateAwareChannelUpstreamHandler {

    @Override
    public void channelIdle(ChannelHandlerContext ctx,
        IdleStateEvent e) throws Exception {
        // Shut down connection to client and roll everything back.
    }

    @Override public void channelConnected(ChannelHandlerContext ctx,
        ChannelStateEvent e) throws Exception {
        if (!acquireConnectionSlot()) {
            // Maximum number of allowed server connections reached,
            // respond with 503 service unavailable
            // and shutdown connection.
        } else {
            // Set up the connection's request pipeline.
        }
    }

    @Override public void messageReceived(ChannelHandlerContext ctx,
        MessageEvent e) throws Exception {
        if (isDone()) return;

        if (e.getMessage() instanceof HttpRequest) {
            handleHttpRequest((HttpRequest) e.getMessage());
        } else if (e.getMessage() instanceof HttpChunk) {
            handleHttpChunk((HttpChunk)e.getMessage());
        }
    }
}
```

The gist of Droplr's server request validation

If there's an active handler for the current request and it accepts chunks, it then passes on the chunk.

As explained previously in this book, you should never execute non-CPU-bound code on Netty's I/O threads—you'll be stealing away precious resources from Netty and thus affecting the server's throughput.

For this reason, both the `HttpRequest` and `HttpChunk` may hand off the execution to the request handler by switching over to a different thread. This happens when the request handlers aren't CPU-bound, whether because they access the database or perform logic that's not confined to local memory or CPU.

When thread-switching occurs, it's imperative that all the blocks of code execute in serial fashion; otherwise we'd risk, for an upload, having `HttpChunk` *n-1* being processed after `HttpChunk` *n* and thus corrupting the body of the file. (We'd be swapping how bytes were laid out in the uploaded file.) To cope with this, I created a custom thread-pool executor that ensures all tasks sharing a common identifier will be executed serially.

From here on, the data (requests and chunks) ventures out of the realms of Netty and Droplr.

I'll explain briefly how the request handlers are built for the sake of shedding some light on the bridge between the `RequestController`—which lives in Netty-land—and the handlers—Droplr-land. Who knows, maybe this will help you architect your own server!

The request handlers

Request handlers provide Droplr's functionality. They're the endpoints behind URIs such as /account or /drops. They're the logic cores—the server's interpreters of clients' requests.

Request handler implementations are where the framework actually becomes Droplr's API server.

The parent interface

Each request handler, whether directly or through a subclass hierarchy, is a realization of the interface `RequestHandler`.

In its essence, the `RequestHandler` interface represents a stateless handler for requests (instances of `HttpRequest`) and chunks (instances of `HttpChunk`). It's an extremely simple interface with a couple of methods to help the request controller perform and/or decide how to perform its duties, such as:

- Is the request handler stateful or stateless? Does it need to be cloned from a prototype or can the prototype be used to handle the request?
- Is the request handler CPU or non-CPU bound? Can it execute on Netty's worker threads or should it be executed in a separate thread pool?
- Roll back current changes.
- Clean up any used resources.

This interface is all the `RequestController` knows about actions. Through its very clear and concise interface, the controller can interact with stateful and stateless, CPU-bound and non-CPU-bound handlers (or combinations of these) in an isolated and implementation-agnostic fashion.

Handler implementations

The simplest realization of `RequestHandler` is `AbstractRequestHandler`, which represents the root of a subclass hierarchy that becomes ever more specific until it reaches the actual handlers that provide all of Droplr's functionality. Eventually it leads to the stateful implementation `SimpleHandler`, which executes in a non-IO-worker thread and is therefore not CPU-bound. `SimpleHandler` is ideal for quickly implementing endpoints that do the typical tasks of reading in JSON, hitting the database, and then writing out some JSON.

THE UPLOAD REQUEST HANDLER

The upload request handler is the crux of the whole Droplr API server. It was the action that shaped the design of the `webserver` module—the *frameworky* part of the server—and it's by far the most complex and tuned piece of code in the whole stack.

During uploads, the server has dual behaviors:

- On one side, it acts as a server for the API clients that are uploading the files.
- On the other side, it acts as client to S3 to push the data it receives from the API clients.

To act as a client, the server uses an HTTP client library that is also built with Netty.[1] This asynchronous library exposes an interface that perfectly matches the needs of the server. It begins executing an HTTP request and allows data to be fed to it as it becomes available, and this greatly reduces the complexity of the client facade of the upload request handler.

14.1.5 Performance

After the initial version of the server was complete, I ran a batch of performance tests. The results were nothing short of mind blowing. After continuously increasing the load in disbelief, I saw the new server peak at 10~12x faster uploads over the old LAMP stack—a full order of magnitude faster—and it could handle over 1000x more concurrent uploads, for a total of nearly 10 k concurrent uploads (running on a single EC2 large instance).

The following factors contributed to this:

- It was running in a tuned JVM.
- It was running in a highly tuned custom stack, created specifically to address this problem, instead of an all-purpose web framework.
- The custom stack was built with Netty using NIO (selector-based model), which meant it could scale to tens or even hundreds of thousands of concurrent connections, unlike the one-process-per-client LAMP stack.
- There was no longer the overhead of receiving a full file and then uploading it to S3 in two separate phases. The file was now streamed directly to S3.

[1] You can find the HTTP client library at https://github.com/brunodecarvalho/http-client.

- Because the server was now streaming files,
  - It was not spending time on I/O operations, writing to temporary files and later reading them in the second stage of the upload.
  - It was using less memory for each upload, which meant more parallel uploads could take place.
- Thumbnail generation became an asynchronous post-process.

14.1.6 Summary—standing on the shoulders of giants

All of this was possible thanks to Netty's incredibly well-designed API and performant nonblocking I/O architecture.

Since the launch of Droplr 2.0 in December 2011, we've had virtually zero downtime at the API level. A couple of months ago we interrupted a year-and-a-half clean run of 100% infrastructure uptime due to a scheduled full-stack upgrade (databases, OS, major server and daemons codebase upgrade) that took just under an hour.

The servers soldier on, day after day, taking hundreds—sometimes thousands—of concurrent requests per second, all the while keeping both memory and CPU use to levels so low it's hard to believe they're actually doing such an incredible amount of work:

- CPU use rarely ever goes above 5%.
- Memory footprint can't be accurately described as the process starts with 1 GB of preallocated memory, with the JVM configured to grow up to 2 GB if necessary, and not a single time in the past two years has this happened.

Anyone can throw more machines at any given problem, but Netty helped Droplr scale intelligently, and keep the server bills pretty low.

14.2 Firebase—a real-time data synchronization service

Sara Robinson, VP of Developer Happiness

Greg Soltis, VP of Cloud Architecture

Real-time updates are an integral part of the user experience in modern applications. As users come to expect this behavior, more and more applications are pushing data changes to users in real time. Real-time data synchronization is difficult to achieve with the traditional three-tiered architecture, which requires developers to manage their own ops, servers, and scaling. By maintaining real-time, bidirectional communication with the client, Firebase provides an immediately intuitive experience allowing developers to synchronize application data across diverse clients in a few minutes—all without any backend work, servers, ops, or scaling required.

Implementing this presented a difficult technical challenge, and Netty was the optimal solution in building the underlying framework for all network communications in Firebase. This study will provide an overview of Firebase's architecture, and then examine three ways Firebase uses Netty to power its real-time synchronization service:

- Long polling
- HTTP 1.1 keep-alive and pipelining
- Control of SSL handler

14.2.1 The Firebase architecture

Firebase allows developers to get an application up and running using a two-tiered architecture. Developers simply include the Firebase library and write client-side code. The data is exposed to the developer's code as JSON and is cached locally. The library handles synchronizing this local cache with the master copy, which is stored on Firebase's servers. Changes made to any data are synchronized in real time to potentially hundreds of thousands of clients connected to Firebase. The interaction between multiple clients across platforms and devices and Firebase is depicted in figure 14.3.

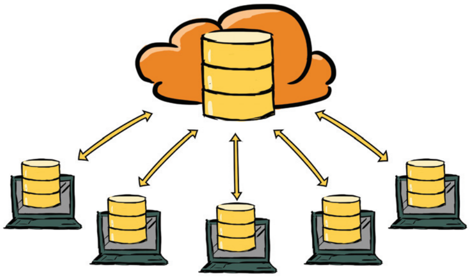


Figure 14.3 Firebase architecture

Firebase servers take incoming data updates and immediately synchronize them to all of the connected clients that have registered interest in the changed data. To enable real-time notification of state changes, clients maintain an active connection to Firebase at all times. This connection may range from an abstraction over a single Netty channel to an abstraction over multiple channels or even multiple, concurrent abstractions if the client is in the middle of switching transport types.

Because clients can connect to Firebase in a variety of ways, it's important to keep the connection code modular. Netty's `Channel` abstraction is a fantastic building block for integrating new transports into Firebase. In addition, the pipeline-and-handler pattern makes it simple to keep transport-specific details isolated and provide a common message stream abstraction to the application code. Similarly, this greatly simplifies adding support for new protocols. Firebase added support for a binary transport simply by adding a few new handlers to the pipeline. Netty's speed, level of abstraction, and fine-grained control made it an excellent framework for implementing real-time connections between the client and server.

14.2.2 Long polling

Firebase uses both long polling and WebSocket transports. The long-polling transport is highly reliable across all browsers, networks, and carriers; the WebSocket-based transport is faster but not always available due to limitations of browsers/clients. Initially, Firebase connects using long polling and then upgrades to WebSockets if possible. For the minority of Firebase traffic that doesn't support WebSockets, Firebase uses Netty to implement a custom library for long polling tuned to be highly performant and responsive.

The Firebase library logic deals with bidirectional streams of messages with notifications when either side closes the stream. Although this is relatively simple to implement on top of TCP or WebSockets, it presents a challenge when dealing with a long-polling transport. The two properties that must be enforced for the long-polling case are

- Guaranteed in-order delivery of messages
- Close notifications

Guaranteed in-order delivery of messages

In-order delivery for long polling can be achieved by having only a single request outstanding at a given time. Because the client won't send another request until it receives a response from its last request, it can guarantee that its previous messages were received and that it's safe to send more. Similarly, on the server side, there won't be a new request outstanding until the client has received the previous response. Therefore, it's always safe to send everything that's buffered up in between requests. However, this leads to a major drawback. Using the single-request technique, both the client and server spend a significant amount of time buffering up messages. If the client has new data to send but already has an outstanding request, for example, it must wait for the server to respond before sending the new request. This could take a long time if there's no data available on the server.

A more performant solution is to tolerate more requests being in flight concurrently. In practice, this can be achieved by swapping the single-request pattern for the at-most-two-requests pattern. This algorithm has two parts:

- Whenever a client has new data to send, it sends a new request unless two are already in flight.
- Whenever the server receives a request from a client, if it already has an open request from the client, it immediately responds to the first even if there is no data.

This provides an important improvement over the single-request pattern: both the client's and server's buffer time are bound to at most a single network round-trip.

Of course, this increase in performance doesn't come without a price; it results in a commensurate increase in code complexity. The long-polling algorithm no longer guarantees in-order delivery, but a few ideas from TCP can ensure that messages are delivered in order. Each request sent by the client includes a serial number, incremented for each request. In addition, each request includes metadata about the number of messages in the payload. If a message spans multiple requests, the portion of the message contained in this payload is included in the metadata.

The server maintains a ring buffer of incoming message segments and processes them as soon as they're complete and no incomplete messages are ahead of them. Downstream is easier because the long-polling transport responds to an HTTP `GET` request and doesn't have the same restrictions on payload size. In this case, a serial number is included and is incremented once for each response. The client can process all messages in the list as long as it has received all responses up to the given serial number. If it hasn't, it buffers the list until it receives the outstanding responses.

CLOSE NOTIFICATIONS

The second property enforced in the long-polling transport is close notification. In this case, having the server be aware that the transport has closed is significantly more important than having the client recognize the close. The Firebase library used by clients queues up operations to be run when a disconnect occurs, and those operations can have an impact on other still-connected clients. So it's important to know when a client has actually gone away. Implementing a server-initiated close is relatively simple and can be achieved by responding to the next request with a special protocol-level close message.

Implementing client-side close notifications is trickier. The same close notification can be used, but there are two things that can cause this to fail: the user can close the browser tab, or the network connection could disappear. The tab-closure case is handled with an `iframe` that fires a request containing the close message on page unload. The second case is dealt with via a server-side timeout. It's important to pick your timeout values carefully, because the server is unable to distinguish a slow network from a disconnected client. That is to say, there's no way for the server to know that a request was actually delayed for a minute, rather than the client losing its network connection. It's important to choose an appropriate timeout that balances the cost of false positives (closing transports for clients on slow networks) against how quickly the application needs to be aware of disconnected clients.

Figure 14.4 demonstrates how the Firebase long-polling transport handles different types of requests.

In this diagram, each long-poll request indicates different types of scenarios. Initially, the client sends a poll (poll 0) to the server. Some time later, the server receives data from elsewhere in the system that is destined for this client, so it responds to poll 0 with the data. As soon as the poll returns, the client sends a new poll (poll 1), because it currently has none outstanding. A short time later, the client needs to send data to the server. Since it only has a single poll outstanding, it sends a new one (poll 2) that includes the data to be delivered. Per the protocol, as soon as the server has two simultaneous polls from the same client, it responds to the first one. In this case, the server has no data available for the client, so it sends back an empty response. The client also maintains a timeout and will send a second poll when it fires, even if it has no

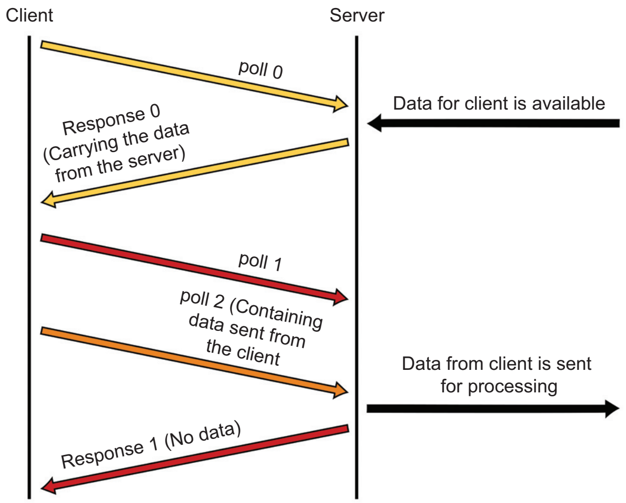


Figure 14.4 Long polling

additional data to send. This insulates the system from failures due to browsers timing out slow requests.

14.2.3 HTTP 1.1 keep-alive and pipelining

With HTTP 1.1 keep-alive, multiple requests can be sent on one connection to a server. This allows for pipelining—new requests can be sent without waiting for a response from the server. Implementing support for pipelining and keep-alive is typically straightforward, but it gets significantly more complex when mixed with long polling.

If a long-polling request is immediately followed by a REST (Representational State Transfer) request, there are some considerations that need to be taken into account to ensure the browser performs properly. A channel may mix asynchronous messages (long-poll requests) with synchronous messages (REST requests). When a synchronous request comes in on one channel, Firebase must synchronously respond to all preceding requests in that channel in order. For example, if there's an outstanding long-poll request, the long-polling transport needs to respond with a no-op before handling the REST request.

Figure 14.5 illustrates how Netty lets Firebase respond to multiple request types in one socket.

If the browser has more than one connection open and is using long polling, it will reuse the connection for messages from both of those open

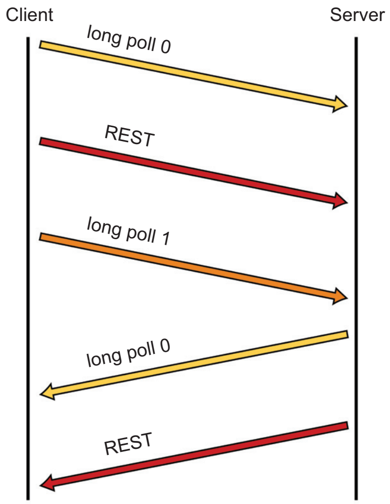


Figure 14.5 Network diagram

tabs. Given long-polling requests, this is difficult and requires proper management of a queue of HTTP requests. Long-polling requests can be interrupted, but proxied requests can't. Netty made serving multiple request types easy:

- *Static* HTML *pages*—Cached content that can be returned with no processing; examples include a single-page HTML app, robots.txt, and crossdomain.xml.
- REST *requests*—Firebase supports traditional `GET`, `POST`, `PUT`, `DELETE`, `PATCH`, and `OPTIONS` requests.
- *WebSocket*—A bidirectional connection between a browser and a Firebase server with its own framing protocol.
- *Long polling*—These are similar to HTTP `GET` requests but are treated differently by the application.
- *Proxied requests*—Some requests can't be handled by the server that receives them. In that case, Firebase proxies the request to the correct server in its cluster, so that end users don't have to worry about where data is located. These are like the REST requests, but the proxying server treats them differently.
- *Raw bytes over* SSL—A simple TCP socket running Firebase's own framing protocol and optimized handshaking.

Firebase uses Netty to set up its pipeline to decode an incoming request and then reconfigure the remainder of the pipeline appropriately. In some cases, like WebSockets and raw bytes, once a particular type of request has been assigned a channel, it will stay that way for its entire duration. In other cases, like the various HTTP requests, the assignment must be made on a per-message basis. The same channel could handle REST requests, long-polling requests, and proxied requests.

14.2.4 Control of SslHandler

Netty's `SslHandler` class is an example of how Firebase uses Netty for fine-grained control of its network communications. When a traditional web stack uses an HTTP server like Apache or Nginx to pass requests to the app, incoming SSL requests have already been decoded when they're received by the application code. With a multitenant architecture, it's difficult to assign portions of the encrypted traffic to the tenant of the application using a specific service. This is complicated by the fact that multiple applications could use the same encrypted channel to talk to Firebase (for instance, the user might have two Firebase applications open in different tabs). To solve this, Firebase needs enough control in handling SSL requests before they are decoded.

Firebase charges customers based on bandwidth. However, the account to be charged for a message is typically not available before the SSL decryption has been performed, because it's contained in the encrypted payload. Netty allows Firebase to intercept traffic at multiple points in the pipeline, so the counting of bytes can start as soon as byes come in off the wire. After the message has been decrypted and processed by Firebase's server-side logic, the byte count can be assigned to the appropriate account. In building this feature, Netty provided control for handling network communications at every layer of the protocol stack, and also allowed for very accurate billing, throttling, and rate limiting, all of which had significant business implications.

Netty made it possible to intercept all inbound and outbound messages and to count bytes with a small amount of Scala code.

Listing 14.3 Setting up the `ChannelPipeline`

```
case class NamespaceTag(namespace: String)

class NamespaceBandwidthHandler extends ChannelDuplexHandler {
    private var rxBytes: Long = 0
    private var txBytes: Long = 0
    private var nsStats: Option[NamespaceStats] = None

    override def channelRead(ctx: ChannelHandlerContext, msg: Object) {
        msg match {
            case buf: ByteBuf => {
                rxBytes += buf.readableBytes(
                                      tryFlush(ctx)
            }
            case _ => { }
        }
        super.channelRead(ctx, msg)
    }

     override def write(ctx: ChannelHandlerContext, msg: Object,
             promise: ChannelPromise) {
         msg match {
             case buf: ByteBuf => {
                 txBytes += buf.readableBytes()
                 tryFlush(ctx)
                 super.write(ctx, msg, promise)
             }
             case tag: NamespaceTag => {
                 updateTag(tag.namespace, ctx)
             }
             case _ => {
                 super.write(ctx, msg, promise)
             }
         }
     }

     private def tryFlush(ctx: ChannelHandlerContext) {
         nsStats match {
             case Some(stats: NamespaceStats) => {
                 stats.logOutgoingBytes(txBytes.toInt)
                 txBytes = 0
                 stats.logIncomingBytes(rxBytes.toInt)
                 rxBytes = 0
             }
             case None => {
                 // no-op, we don't have a namespace
             }
         }
    }
```

When a message comes in, counts the number of bytes

When there is an outbound message, counts those bytes as well

If a tag is received, ties this channel to an account, remembers the account, and assigns the current byte counts to it.

If there's already a tag for the namespace the channel belongs to, assigns the bytes to that account and resets the counters

```
    private def updateTag(ns: String, ctx: ChannelHandlerContext) {
         val (_, isLocalNamespace) = NamespaceOwnershipManager.getOwner(ns)
         if (isLocalNamespace) {
             nsStats = NamespaceStatsListManager.get(ns)
             tryFlush(ctx)
         } else {
             // Non-local namespace, just flush the bytes
            txBytes = 0
            rxBytes = 0
         }
     }
}
```

If the count isn't applicable to this machine, ignores it and resets the counters

14.2.5 Firebase summary

Netty plays an indispensable role in the server architecture of Firebase's real-time data synchronization service. It allows support for a heterogeneous client ecosystem, which includes a variety of browsers, along with clients that are completely controlled by Firebase. With Netty, Firebase can handle tens of thousands of messages per second on each server. Netty is especially awesome for several reasons:

- *It's fast.* It took only a few days to develop a prototype, and was never a production bottleneck.
- *It's positioned well in the abstraction layer.* Netty provides fine-grained control where necessary and allows for customization at every step of the control flow.
- *It supports multiple protocols over the same port.* HTTP, WebSockets, long polling, and standalone TCP.
- *Its GitHub repo is top-notch.* Well-written javadocs make it frictionless to develop against.
- *It has a highly active community.* The community is very responsive on issue maintenance and seriously considers all feedback and pull requests. In addition, the team provides great and up-to-date example code. Netty is an excellent, well-maintained framework and it has been essential in building and scaling Firebase's infrastructure. Real-time data synchronization in Firebase wouldn't be possible without Netty's speed, control, abstraction, and extraordinary team.

14.3 Urban Airship—building mobile services

Erik Onnen, Vice President of Architecture

As smartphone use grows across the globe at unprecedented rates, a number of service providers have emerged to assist developers and marketers toward the end of providing amazing end-user experiences. Unlike their feature phone predecessors, smartphones crave IP connectivity and seek it across a number of channels (3G, 4G, WiFi, WiMAX, and Bluetooth). As more and more of these devices access public networks via IP-based protocols, the challenges of scale, latency, and throughput become more and more daunting for back-end service providers.

Thankfully, Netty is well suited to many of the concerns faced by this thundering herd of always-connected mobile devices. This chapter will detail several practical applications of Netty in scaling a mobile developer and marketer platform, Urban Airship.

14.3.1 Basics of mobile messaging

Although marketers have long used SMS as a channel to reach mobile devices, a more recent functionality called *push notifications* is rapidly becoming the preferred mechanism for messaging smartphones. Push notifications commonly use the less expensive data channel and the price per message is a fraction of the cost of SMS. The throughput of push notifications is commonly two to three orders of magnitude higher than SMS, making it an ideal channel for breaking news. Most importantly, push notifications give users device-driven control of the channel. If a user dislikes the messaging from an application, the user can disable notifications for an application or outright delete the application.

At a very high level, the interaction between a device and push notification behavior is similar to the depiction in figure 14.6.

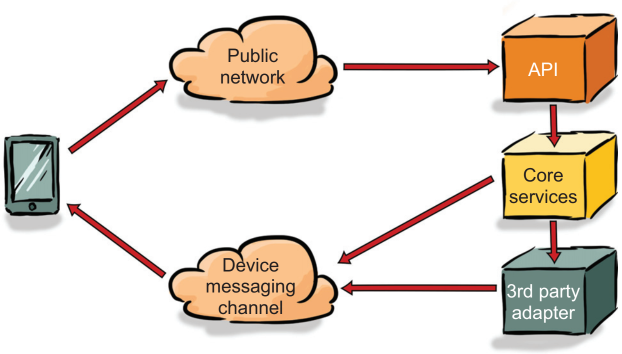


Figure 14.6 High-level mobile messaging platform integration

At a high level, when an application developer wants to send push notifications to a device, the developer must plan to store information about the device and its application installation.[1] Commonly, an application installation will execute code to retrieve a platform-specific identifier and report that identifier back to a centralized service where the identifier is persisted. Later, logic external to the application installation will initiate a request to deliver a message to the device.

Once an application installation has registered its identifier with a back-end service, the delivery of a push message can in turn take two paths. In the first path, a message can be delivered directly to the application itself, with the application maintaining a direct connection to a back-end service. In the second and more common

[1] Some mobile OSes allow a form of push notifications called local notifications that would not follow this approach.

approach, an application will rely on a third party to deliver the message to the application on behalf of a back-end service. At Urban Airship, both approaches to delivering push notifications are used, and both leverage Netty extensively.

14.3.2 Third-party delivery

In the case of third-party push delivery, every push notification platform provides a different API for developers to deliver messages to application installations. These APIs differ in terms of their protocol (binary vs. text), authentication (OAuth, X.509, and so on), and capabilities. Each approach has its own unique challenges for integration as well as for achieving optimal throughput.

Despite the fact that the fundamental purpose of each of these providers is to deliver a notification to an application, each takes a different approach with significant implications for system integrators. For example, Apple Push Notification Service (APNS) defines a strictly binary protocol; other providers base their service on some form of HTTP, all with subtle variations that affect how to best achieve maximum throughput. Thankfully, Netty is an amazingly flexible tool and it significantly helps smoothing over the differences between the various protocols.

The following sections will provide examples of how Urban Airship uses Netty to integrate with two of the listed providers.

14.3.3 Binary protocol example

Apple's APNS is a binary protocol with a specific, network byte-ordered payload. Sending an APNS notification involves the following sequence of events:

1. Connect a TCP socket to APNS servers over an SSLv3 connection, authenticated with an X.509 certificate.
2. Format a binary representation of a push message structured according to the format defined by Apple.[1]
3. Write the message to the socket.
4. Read from the socket if you're ready to determine any error codes associated with a sent message.
5. In the case of an error, reconnect the socket and continue from step 2.

As part of formatting the binary message, the producer of the message is required to generate an identifier that's opaque to the APNS system. In the event of an invalid message (incorrect formatting, size, or device information, for example), the identifier will be returned to the client in the error response message of step 4.

[1] For information on APNS: http://docs.aws.amazon.com/sns/latest/dg/mobile-push-apns.html, http://bit.ly/189mmpG.

At face value the protocol seems straightforward, but there are nuances to successfully addressing all of the preceding concerns, in particular on the JVM:

- The APNS specification dictates that certain payload values should be sent in big-endian ordering (for example, token length).
- Step 3 in the previous sequence requires one of two solutions. Because the JVM will not allow reading from a closed socket even if data exists in the output buffer, you have two options:
  - After a write, perform a blocking read with a timeout on the socket. This has multiple disadvantages:
    - The amount of time to block waiting for an error is non-deterministic. An error may occur in milliseconds or seconds.
    - As socket objects can't be shared across multiple threads, writes to the socket must immediately block while waiting for errors. This has dramatic implications for throughput. If a single message is delivered in a socket write, no additional messages can go out on that socket until the read timeout has occurred. When you're delivering tens of millions of messages, a three-second delay between messages isn't acceptable.
    - Relying on a socket timeout is an expensive operation. It results in an exception being thrown and several unnecessary system calls.
  - Use asynchronous I/O. In this model, neither reads nor writes block. This allows writers to continue sending messages to APNS while at the same time allowing the OS to inform user code when data is ready to be read.

Netty makes addressing all of these concerns trivial while at the same time delivering amazing throughput.

First, let's see how Netty simplifies packing a binary APNS message with correct endian ordering.

Listing 14.4 `ApnsMessage` implementation

```
public final class ApnsMessage {
    private static final byte COMMAND = (byte) 1;
    public ByteBuf toBuffer() {
        short size = (short) (1 + // Command
            4 + // Identifier
            4 + // Expiry
            2 + // DT length header
            32 + //DS length
            2 + // body length header
            body.length);

        ByteBuf buf = Unpooled.buffer(size).order(ByteOrder.BIG_ENDIAN);
        buf.writeByte(COMMAND);
        buf.writeInt(identifier);
        buf.writeInt(expiryTime);
```

An APNS message always starts with a command 1 byte in size, so that value is coded as a constant.

Messages size varies, so for efficiency it is calculated before the ByteBuf is created.

At creation the ByteBuf is sized exactly and the endianness for APNS is specified.

Various values are inserted into the buffer from state maintained elsewhere in the class.

```
        buf.writeShort((short) deviceToken.length);
        buf.writeBytes(deviceToken);
        buf.writeShort((short) body.length);
        buf.writeBytes(body);
        return buf;
    }
}
```

❶ The deviceToken field in this class (not shown) is a Java byte[].

❷ When the buffer is ready, it is simply returned.

Some important notes on the implementation:

❶ The length property of a Java array is always an integer. However, the APNS protocol requires a 2-byte value. In this case, the length of the payload has been validated elsewhere, so casting to a `short` is safe at this location. Note that without explicitly constructing the `ByteBuf` to be big endian, subtle bugs could occur with values of types `short` and `int`.

❷ Unlike the standard `java.nio.ByteBuffer`, it's not necessary to flip the buffer and worry about its position—Netty's `ByteBuf` handles read and write position management automatically.

In a small amount of code, Netty has made trivial the act of creating a properly formatted APNS message. Because this message is now packed into a `ByteBuf`, it can easily be written directly to a `Channel` connected to APNS when the message is ready for sending.

Connecting to APNS can be accomplished via multiple mechanisms, but at its most basic, a `ChannelInitializer` that populates the `ChannelPipeline` with an `SslHandler` and a decoder is required.

Listing 14.5 Setting up the `ChannelPipeline`

```
public final class ApnsClientPipelineInitializer
    extends ChannelInitializer<Channel> {
    private final SSLEngine clientEngine;

    public ApnsClientPipelineFactory(SSLEngine engine) {
        this.clientEngine = engine;
    }

    @Override
    public void initChannel(Channel channel) throws Exception {
        final ChannelPipeline pipeline = channel.pipeline();
        final SslHandler handler = new SslHandler(clientEngine);
        handler.setEnableRenegotiation(true);
        pipeline.addLast("ssl", handler);
        pipeline.addLast("decoder", new ApnsResponseDecoder());
    }
}
```

An X.509 authenticated request requires a javax.net.ssl.SSLEngine instance.

Constructs a Netty SslHandler

APNS will attempt to renegotiate SSL shortly after connection, need to allow renegotiation.

This class extends Netty's ByteToMessageDecoder and handles cases where APNS returns an error code and disconnects.

It's worth noting how easy Netty makes negotiating an X.509 authenticated connection in conjunction with asynchronous I/O. In early prototypes of APNS code at Urban Airship without Netty, negotiating an asynchronous X.509 authenticated connection required over 80 lines of code and a thread pool simply to connect. Netty hides all the complexity of the SSL handshake, the authentication, and most importantly the encryption of cleartext bytes to cipher text and the key renegotiation that comes along with using SSL. These incredibly tedious, error prone, and poorly documented APIs in the JDK are hidden behind three lines of Netty code.

At Urban Airship, Netty plays a role in all connectivity to numerous third-party push notification services including APNS and Google's GCM. In every case, Netty is flexible enough to allow explicit control over exactly how integration takes place from higher-level HTTP connectivity behavior down to basic socket-level settings such as TCP keep-alive and socket buffer sizing.

14.3.4 Direct to device delivery

The previous section provides insight into how Urban Airship integrates with a third party for message delivery. In referring to figure 14.1, note that two paths exist for delivering messages to a device. In addition to delivering messages through a third party, Urban Airship has experience serving directly as a channel for message delivery. In this capacity, individual devices connect directly to Urban Airship's infrastructure, bypassing third-party providers. This approach brings a distinctly different set of challenges:

- *Socket connections from mobile devices are often short-lived.* Mobile devices frequently switch between different types of networks depending on various conditions. To back-end providers of mobile services, devices constantly reconnect and experience short but frequent periods of connectivity.
- *Connectivity across platforms is irregular.* From a network perspective, tablet devices tend to behave differently than mobile phones, and mobile phones behave differently than desktop computers.
- *Frequency of mobile phone updates to back-end providers is certain to increase.* Mobile phones are increasingly used for daily tasks, producing significant amounts of general network traffic but also analytics data for back-end providers.
- *Battery and bandwidth can't be ignored.* Unlike a traditional desktop environment, mobile phones tend to operate on limited data plans. Service providers must honor the fact that end users have limited battery life and they use expensive, limited bandwidth. Abuse of either will frequently result in the uninstallation of an application, the worst possible outcome for a mobile developer.
- *All aspects of infrastructure will need to scale massively.* As mobile device popularity increases, more application installations result in more connections to a mobile services infrastructure. Each of the previous elements in this list are further complicated by the sheer scale and growth of mobile devices.

Over time, Urban Airship learned several critical lessons as connections from mobile devices continued to grow:

- The diversity of mobile carriers can have a dramatic effect on device connectivity.
- Many carriers don't allow TCP keep-alive functionality. Given that, many carriers will aggressively cull idle TCP sessions.
- UDP isn't a viable channel for messaging to mobile devices because many carriers disallow it.
- The overhead of SSLv3 is an acute pain for short-lived connections.

Given the challenges of mobile growth and the lessons learned by Urban Airship, Netty was a natural fit for implementing a mobile messaging platform for reasons highlighted in the following sections.

14.3.5 Netty excels at managing large numbers of concurrent connections

As mentioned in the previous section, Netty makes supporting asynchronous I/O on the JVM trivial. Because Netty operates on the JVM, and because the JVM on Linux ultimately uses the Linux epoll facility to manage interest in socket file descriptors, Netty makes it possible to accommodate the rapid growth of mobile by allowing developers to easily accept large numbers of open sockets—close to 1 million TCP connections per single Linux process. At numbers of this scale, service providers can keep costs low, allowing a large number of devices to connect to a single process on a physical server.[1]

In controlled testing and with configuration options optimized to use small amounts of memory, a Netty-based service was able to accommodate slightly less than 1 million connections (approximately 998,000). In this case, the limit was fundamentally the Linux kernel imposing a hard-coded limit of 1 million file handles per process. Had the JVM itself not held a number of sockets and file descriptors for JAR files, the server would likely have been capable of handling even more connections, all on a 4 GB heap. Leveraging this efficiency, Urban Airship has successfully sustained over 20 million persistent TCP socket connections to its infrastructure for message delivery, all on a handful of servers.

It's worth noting that while in practice a single Netty-based service is capable of handling nearly a million inbound TCP socket connections, doing so is not necessarily pragmatic or advisable. As with all things in distributed computing, hosts will fail, processes will need to be restarted, and unexpected behavior will occur. As a result of these realities, proper capacity planning means considering the consequences of a single process failing.

[1] Note the distinction of a *physical server* in this case. Although virtualization offers many benefits, leading cloud providers were regularly unable to accommodate more than 200,000–300,000 concurrent TCP connections to a single virtual host. With connections at or above this scale, expect to use bare metal servers and expect to pay close attention to the NIC (Network Interface Card) vendor.

14.3.6 Summary—Beyond the perimeter of the firewall

We've demonstrated two everyday uses of Netty at the perimeter of the Urban Airship network. Netty works exceptionally well for these purposes, but it has also found a home as scaffolding for many other components inside Urban Airship.

Internal RPC framework

Netty has been the centerpiece of an internal RPC framework that has consistently evolved inside Urban Airship. Today, this framework processes hundreds of thousands of requests per second with very low latency and exceptional throughput. Nearly every API request fielded by Urban Airship processes through multiple back-end services with Netty at the core of all of those services.

Load and performance testing

Netty has been used at Urban Airship for several different load- and performance-testing frameworks. For example, to simulate millions of device connections in testing the previously described device-messaging service, Netty was used in conjunction with a Redis (http://redis.io/) instance to test end-to-end message throughput with a minimal client-side footprint.

Asynchronous clients for commonly synchronous protocols

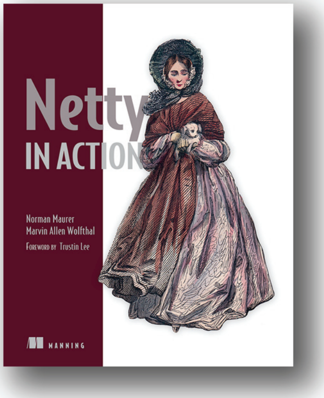
For some internal uses, Urban Airship has been experimenting with Netty to create asynchronous clients for typically synchronous protocols, including services like Apache Kafka (http://kafka.apache.org/) and Memcached (http://memcached.org/). Netty's flexibility easily allows us to craft clients that are asynchronous in nature but that can be converted back and forth between truly asynchronous or synchronous implementations without requiring upstream code changes.

All in all, Netty has been a cornerstone of Urban Airship as a service. The authors and community are fantastic and have produced a truly first-class framework for anything requiring networking on the JVM.

14.4 Summary

This chapter aimed at providing insight into real-world use of Netty and how it has helped companies to solve significant networking problems. It's worth noting how in all cases Netty was leveraged not only as a code framework, but also as an essential component of development and architectural best practices.

In the next chapter we'll present case studies contributed by Facebook and Twitter describing open source projects that evolved from Netty-based code originally developed to address internal needs.



Netty is a Java-based networking framework that manages complex networking, multithreading, and concurrency for your applications. And Netty hides the boilerplate and low-level code, keeping your business logic separate and easier to reuse. With Netty, you get an easy-to-use API, leaving you free to focus on what's unique to your application.

Netty in Action introduces the Netty framework and shows you how to incorporate it into your Java network applications. You will discover how to write highly scalable applications without getting into low-level APIs. The book teaches you to think in an asynchronous way as you work through its many hands-on examples and helps you master the best practices of building large-scale network apps.

What's inside

- Netty from the ground up
- Asynchronous, event-driven programming
- Implementing services using different protocols
- Covers Netty 4.x

This book assumes readers are comfortable with Java and basic network architecture.

index