



BUILDING APPLICATIONS IN THE CLOUD

Concepts, Patterns, and Projects

CHRISTOPHER M. MOYER

Building Applications in the Cloud

This page intentionally left blank

Building Applications in the Cloud

Concepts, Patterns,
and Projects

Christopher M. Moyer

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Howard, Ken, 1962-

Individuals and interactions : an agile guide / Ken Howard,
Barry Rogers.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-321-71409-1 (pbk. : alk. paper) 1. Teams in the
workplace. 2. Communication. I. Rogers, Barry, 1963- II. Title.

HD66.H695 2011

658.4'022 -- dc22

2011001898

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447
ISBN-13: 978-0-321-72020-7
ISBN-10: 0-321-72020-2

Text printed in the United States on recycled paper at R.R.
Donnelley & Sons, Crawfordsville, Indiana.

First printing April 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Senior Project Editor
Lori Lyons

Copy Editor
Apostrophe Editing
Services

Indexer
Ken Johnson

Proofreader
Sheri Cain

Technical Reviewers
Kevin Davis
Mocky Habeeb
Colin Percival

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Cover Illustrator
Lynn A. Moyer
www.designbylynn.com

Compositor
Nonie Ratcliff



*To my wonderful wife Lynn,
without whom this book would never
have been finished.*



This page intentionally left blank

Table of Contents

Preface xv

About the Author xx

Introduction 1

What Is Cloud Computing? 1

The Evolution of Cloud Computing 2

 The Main Frame 3

 The PC Revolution 4

 The Fast Internet 5

 The Cloud 6

 HTML5 and Local Storage 7

 The Dawn of Mobile Devices 9

Threading, Parallel Processing, and
Parallel Computing 10

How Does Cloud-Based Development Differ
from Other Application Development? 11

What to Avoid 13

Getting Started in the Cloud 14

 Selecting a Cloud Pattern 16

 Implementing a Cloud Pattern 17

I: Concepts

1 Fundamentals of Cloud Services 21

Origins of Cloud Computing 21

What Is a Cloud Service? 23

 Compute 24

 Storage 24

 Connectivity 24

The Legacy Pattern	25
Just Because It's in the Cloud Doesn't Mean It Scales	26
Failure as a Feature	27
Consistency, Availability, and Partition Tolerance	28
Consistency	29
Availability	30
Partition Tolerance	30
Eventual Consistency	31
Summary	32

2 Making Software a Service 33

Tools Used in This Book	34
Signing Up for Amazon Web Services	34
Installing boto	35
Setting Up the Environment	36
Testing It All	38
What Does Your Application Need?	39
Data Layer	40
Introducing the AWS Databases	41
Application Layer	47
Using Elastic Load Balancing	47
Adding Servers to the Load Balancer	49
Automatically Registering an Instance with a Load Balancer	51
HTTP and REST	53
The Header	53
The Body	57
Methods	58
Authorization Layer	62
Client Layer	64
Browser-Based Clients	65
Native Applications	66
Summary	67

3 Cloud Service Providers 69

Amazon Web Services	69
Simple Storage Service (S3)	71
CloudFront	77
Simple Queue Service (SQS)	80
Elastic Compute Cloud (EC2)	83
Elastic Block Storage (EBS)	88
Elastic Load Balancing (ELB)	91
SimpleDB	93
Relational Database Service (RDS)	95
Simple Notification Service (SNS)	102
Virtual Private Cloud (VPC)	106
Google Cloud	108
AppEngine	108
Google Storage	111
Rackspace Cloud	112
CloudFiles	112
CloudServers	113
CloudSites	113
Summary	114

II: Patterns

4 Designing an Image 117

Prepackaged Images	119
Overview	119
Reasons for Usage	119
Description	120
Implementation	120
Example	122
Summary	125
Singleton Instances	127
Overview	127
Reasons for Usage	127
Description	128
Implementation	128

Example	128
Summary	130
Prototype Images	131
Overview	131
Reasons for Usage	131
Description	132
Implementation	132
Example	133
Summary	135
5 Designing an Architecture	137
Adapters	139
Overview	139
Reasons for Usage	140
Description	140
Implementation	140
Example	141
Summary	146
Facades	147
Overview	147
Reasons for Usage	148
Description	148
Implementation	148
Example	149
Summary	152
Proxies and Balancers	153
Overview	153
Reasons for Usage	153
Description	154
Implementation	154
Example	155
Summary	158
6 Executing Actions on Data	159
Queuing	161
Overview	161
Reasons for Usage	162

Description	162
Implementation	163
Example	163
Summary	170
Command	173
Overview	173
Reasons for Usage	173
Description	174
Implementation	174
Example	175
Summary	179
Iterator	181
Overview	181
Reasons for Usage	181
Description	182
Implementation	182
Example	183
Summary	185
Observer	187
Overview	187
Reasons for Usage	188
Description	188
Implementation	188
Example	189
Summary	191
7 Clustering	193
The n-Tier Web Pattern	195
Overview	195
Reasons for Usage	196
Description	196
Implementation	197
Example	198
Summary	210
Semaphores and Locking	211
Overview	211
Reasons for Usage	211

Description	212
Implementation	212
Example	213
Summary	218
Map/Reduce	219
Overview	219
Reasons for Usage	220
Description	220
Implementation	220
Example	222
Summary	226

III: Projects

8 A Simple Weblog 229

Storage	229
Creating a Domain	231
The User Object	232
The Post Object	234
The Comment Object	237
Application	240
A Brief Introduction to WSGI	241
The DB Handler	243
The User, Post, and Comment Handlers	248
Spam Classification	249
Presentation	253
Setting Up the HTTP Proxy	254
Posts	255
Comments	266
Deploying	272
Starting the Base Instance	272
Installing the Software	273
Installing the Application	275
Installing Apache	276
Bundling the Image	277
Creating the Proxy	279
Summary	279

9 A Weblog Using Marajo	281
Initializing the Environment	282
handlers	283
resources	283
static	283
templates	283
Creating the Resources	284
Creating the Handlers	286
Configuring the Application	287
Creating the Templates	288
Running the Application	289
Creating Custom Templates	289
The List Template	289
The Edit Template	292
Summary	296
Glossary	297
Index	307

This page intentionally left blank

Preface

After a few months working as a developer in a small start-up company migrating existing services to the cloud, I started realizing that there was way too much work to be done just by myself. I started looking around for other developers like myself that could assist me, or replace me if I were to find a better and more exciting opportunity elsewhere. I quickly realized that there are so few people that actually fully comprehend the level of complexity it requires to develop a cloud-based application, and almost all these people were happy with their current companies.

I began to create a series of blog posts about working with cloud-based platforms, still available at <http://blog.coredumped.org>, but soon realized that I could quite literally spend an entire year writing up everything there is to know. This documentation would be better placed in a reference book than simply scattered throughout several blog posts, so I decided to write this book.

The Purpose of This Book

This book isn't designed as a tutorial to be read through from cover to cover. It's not a guide for how to build an application for the cloud, but instead it's designed as a reference point for when you have specific questions. When your boss hands you a new project and tells you to make it scale, check the patterns discussed in this book to see what fits. When you work on a project and you find a specific problem that you don't know how to handle, pick up this book. If you're trying to start on a new project, and you have a perfect idea, but you don't know how to scale it, pick up this book. If you're trying to modify an existing project to scale in the cloud, pick up this book. If you don't know what kinds of

applications you can build with cloud computing, pick up this book.

This book doesn't invent many new patterns but simply shows you the tricks and new techniques that you need to consider while running them in the cloud. Although you can use any patterns discussed in this book in any sort of clustering environment, they're designed to take full advantage of the services provided by cloud computing.

How This Book Should Be Used

This book is divided into three parts. Everyone should read Part I, "Concepts," for a basic understanding of cloud computing. In Part II, "Patterns," you can skip to the patterns you're most interested in. If you've never developed any sort of cloud-based application, you may want to go over the example applications in Part III, "Projects," so that you can see exactly what kinds of applications are best suited for this type of system.

Part I, "Concepts"

Part I is designed to give you a general concept of how to develop in the cloud. It's designed to be read from start to finish and is broken into different key chapters important to development:

- Chapter 1, "Fundamentals of Cloud Services"—Provides a basic set of fundamental ideals when working with cloud-based solutions. This is an absolute must read for any developer beginning with this book.
- Chapter 2, "Making Software a Service"—Provides a basic set of instructions for providing Software as a Service (SaaS). It includes details on why this is a good idea and some basics as to how to properly construct your SaaS.
- Chapter 3, "Cloud Service Providers"—Provides some specific examples of services offered by cloud providers.

Part II, “Patterns”

Part II functions more like a reference manual and provides you with a problem and the pattern that solves that problem:

- Chapter 4, “Designing an Image”—Includes basic patterns for use in building your basic image that is the basis for the rest of your application.
- Chapter 5, “Designing an Architecture”—Includes the patterns used for interacting with external systems, not systems offered by your cloud provider.
- Chapter 6, “Executing Actions on Data”—Includes the patterns used to execute code segments against your data.
- Chapter 7, “Clustering”—Includes the patterns used within a basic framework designed to take advantage of multiserver deployments.

Part III, “Projects”

Part III includes examples of real-world applications of the patterns provided throughout this book. These chapters use the same overall hello world tutorial, but in two different ways:

- Chapter 8, “A Simple Weblog”—Details how to build a simple weblog from scratch, not using any existing frameworks.
- Chapter 9, “A Weblog Using Marajo”—Details how to build a weblog using the Marajo cloud-based Web framework.

Conventions Used in This Book

Throughout this book you occasionally see bold words. These words represent a new term, followed by the definition. If you find a term in the book that you don’t know, see the Glossary for a full listing of definitions.

Words listed in *italic* highlight key important ideas to take away from the section. These are usually used to highlight important keywords in a topic, so if you’re skimming over a section looking for something specific, this should help you find exactly what you need.

Where to Begin

The first question for most people now is, where do you start? How do you quickly begin developing applications? What if you don't want to go through and read about all these things you could do and simply want to get into the meat of how things work?

By picking up this book, you're already on the right track. You already know that you can't simply go to a cloud provider and start launching servers and expect to get exactly what you want out of them. People who just pick up a cloud provider and don't do enough research beforehand typically end up with lots of problems, and usually end up blaming the cloud provider for those problems. This is like buying a stick-shift car without first knowing how to drive it and then complaining to the dealership for selling it to you. If you don't first do some research and preparation, you shouldn't be surprised when you have problems with the cloud. If you're not a developer, you probably would be better suited to using a third party to manage your cloud; but if you're reading this book, I'm going to assume that you're interested in more than just "let that guy handle it."

Acknowledgments

I'd like to thank my peer and mentor Mitch Garnaat for all his help and inspiration to push me to cloud computing. I'd also like to thank the team at Amazon Web Services for pushing the market forward and constantly bringing out new products that make everything in this book possible.

About the Author

Chris Moyer is a recent graduate of RIT, the Rochester Institute of Technology, with a bachelor's degree in Software Engineering. Chris has more than five years experience in programming with a main emphasis on cloud computing. Much of his time has been spent working on the popular *boto* client library, used for communicating with Amazon Web Services. Having studied under the creator of *boto*, Mitch Garnaat, Chris then went on to create two web frameworks based on this client library, known as *Marajo* and *botoweb*. He has also created large scaled applications based on those frameworks.

Chris is currently Vice President of Technology for Newstex, LLC, where he manages the technological development of migrating applications to the cloud, and he also manages his own department, which is actively maintaining and developing several applications. Chris lives with his wife, Lynn, in the New York area.

Introduction

Before diving into how to develop your cloud applications, you need to understand a few key concepts behind cloud computing. The term **cloud computing** has been around for only a few years, but the concepts and patterns behind how to use it have been in use since the dawn of the computing age.

What Is Cloud Computing?

There are literally hundreds of definitions for cloud computing, and most of them make little to no sense at all to anyone other than the people that originally created them. Most companies call their virtual hosting environments **clouds** simply because it connotes power, speed, and scalability. In reality, a cloud is little more than *a cluster of computational and storage resources that has almost limitless expandability*. Most cloud offerings also charge you only by what you use.

The advantage to running your application in a cloud computing environment is typically a lower cost because you have no initial investment for most services, and you don't have to pay for expensive IT staff. Although this is a great reason to switch over to using cloud computing services, it's not the only one. Cloud computing also offers you the advantage of *instantly scaling* any application built using the proper design patterns. Additionally, it offers you the ability to offload your work to the people that have been managing it best for years. It would take a full-trained staff much longer to react to the increased demand because it has to go out and purchase new hardware. In contrast, when you work with a cloud-based platform, you can simply request more hardware usage time from the large pool of available resources. Because these cloud

services are typically provided by larger corporations, they can afford to have the staff available all the time to keep their systems running at peak performance.

If you've ever had a server fail in the middle of the night and have to get it back up and running, then you know how much of a pain it can be to get someone to fix it. By offloading your physical servers to the cloud, you can stop worrying about systems management and start working on your applications.

The goal of any entrepreneur is to build a booming business with millions of customers, but most people don't have enough initial capital to build a server farm that can scale to that sort of potential. If you're just starting up a business, you don't even know if it's going to take off. But say your business does take off suddenly, and now you have to quickly scale your web-based application from handling 20 customers to handling 20,000 customers overnight.

In a traditional environment in which you host your own servers, this could mean you need to not only get a faster pipe out of your server farm, but you also actually need to purchase new or faster servers and build them all up and hope they all work together well. You then need to worry about data management and all sorts of other fun and interesting things that go with maintaining your own cluster. If you had been using a cloud, you'd have simply run a few API calls to your cloud vendor and had 40 new machines behind your application within minutes. In a week if that number drops down to 200 customers, you can terminate those unused instances. You'd have saved money because you didn't have to buy them outright.

The Evolution of Cloud Computing

There's been talk lately about how cloud computing and other new software architectures have been designed similar to how things were designed in the past, and that perhaps we've actually taken a step backward. Although there may be some similarities between how things were done in the past and how things are

done now, the design techniques are actually quite different. Following is a brief history of application development.

The Main Frame

In the beginning, there was the mainframe. The mainframe was a single supercomputer with the fastest processing chips money could buy, and they were a prized commodity. They were so large and took so much cooling and electricity that typically even large businesses wouldn't even have one locally to work with, so you had to communicate with it remotely via a **dumb terminal**, a special computer with almost no resources other than to connect to the mainframe (see Figure I.1). Eventually these mainframe's got smaller, but you were still forced to interact with them via these thin clients.

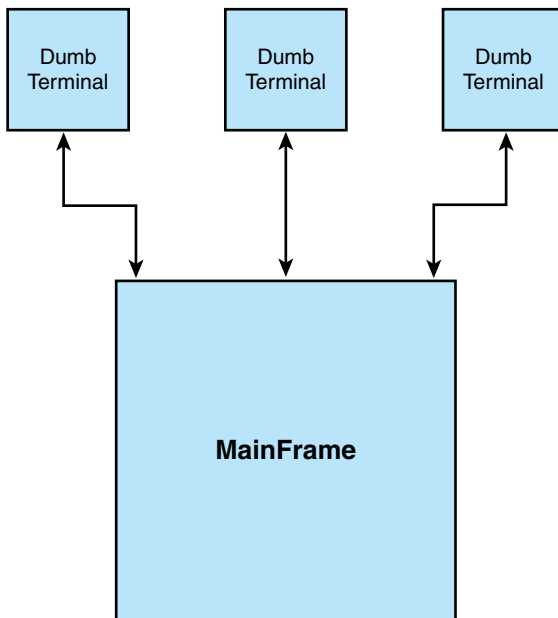


Figure I.1 The mainframe.

These mainframes were designed to run a single process quickly, so there was no need to even think about parallel processing; everything was simply done in sequence. Mainframes needed a complicated scheduling system so that they could allot a given amount of time to any single process but enable others to interject requests in between.

This method of client-server interaction was a huge leap forward from the original system with only one interface to a computer. In many cases, this is still a widely used architecture, although the dumb terminals have been replaced with **thin clients**, which was actually software designed to run on standard computers, not providing much functionality apart from simply connecting to the mainframe. A good example of a thin client still in use today is the modern web browser.

The PC Revolution

As robust as the first mainframes were, they were actually less powerful than a modern digital wrist watch. Eventually technology evolved past the powerful mainframe and into smaller and more powerful devices that were actually housed in an average-sized room. These new devices revolutionized the way software was built, by enabling application developers to run everything locally on the client's system. This meant that your Internet connection speed was no longer a bottleneck and was completely removed from the equation, and the only slowness you would ever see was from your own personal computer. As machines got faster, software continued to demand more from local systems. PCs today, however, are overpowered for most average tasks. This led to the interesting prospect of multitasking, where a single system can be used to run multiple tasks at the same time. Originally, tasking was simply handled by a more advanced version of the scheduler used in the mainframe but eventually became replaced with hardware threading, and even multiprocessor systems.

Many developers refused to adapt to this new multitasking technology, but some did. Those that did developed massively scaled systems that could run in fractions of the time of a single-process

system, taking full advantage of the hardware at hand. This would help aid in the next big leap in technology, the fast Internet.

The Fast Internet

Previously the concern was with network latency and throughput, but eventually the telecommunications industry caught up with the rest of the market. What previously took minutes to send over-the-wire now takes seconds, or even fractions of seconds. With the introduction of the largest infrastructure system ever created, the Internet was born and provided enough throughput and bandwidth to make us rethink how our systems were architected. Many people had come to terms with the idea that software needed to be threaded, but now they took it one step further—developing **clustering**.

Clustering took the idea of processing in parallel to a new level. Instead of simply processing things in parallel on the same machine, clustering is the concept of processing things in parallel on *multiple machines* (see Figure I.2). All these personal computers had much more power than was being used, and most of the time they were entirely idle.

As an example, a few years ago a graphics design company was looking into buying a few servers to run its graphics processing. Graphics manipulation, conversion, and processing is one of the most processor-intensive things required from a computer, so usually it can't be done on local systems without bogging them down or taking a long time. Instead of buying expensive hardware to run and maintain graphics processing, this company decided to take a revolutionary approach.

Its solution was to use the unused processing power on the local systems for graphics processing. It designed a simple queue service that would accept jobs and a simple client interface that ran on every PC in the office that would accept and process jobs only when the computers had downtime. This meant that during off-hours or other periods when employees weren't using their computers, the jobs could be completed, and no additional expensive

hardware was required! This idea of distributed computing across commodity hardware created a new way to develop software applications.

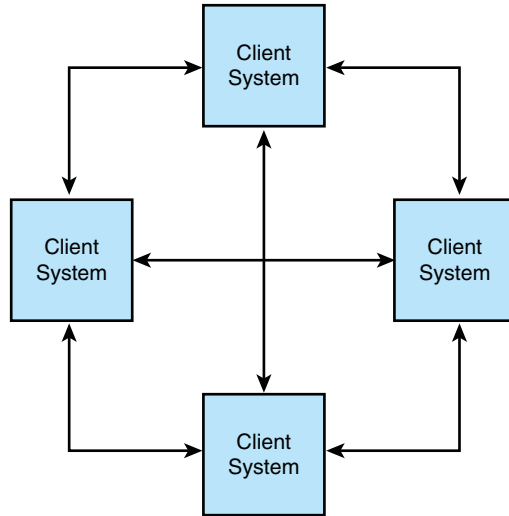


Figure I.2 The cluster system.

The Cloud

Then Amazon released its concept of cloud computing. The idea of a cloud is almost identical to the distributed processing concept, except it uses dedicated systems instead of employees' systems to run the processing jobs (see Figure I.3). This ensures that you'll always have the capacity required to run your jobs. Although it is more expensive (although not much so), you don't have to provide upfront capital to purchase the extra systems; instead you can instantly acquire a few more servers for temporary usage. Again, this revolutionized the way people thought about software and how to design it. However, haven't you seen this approach before?

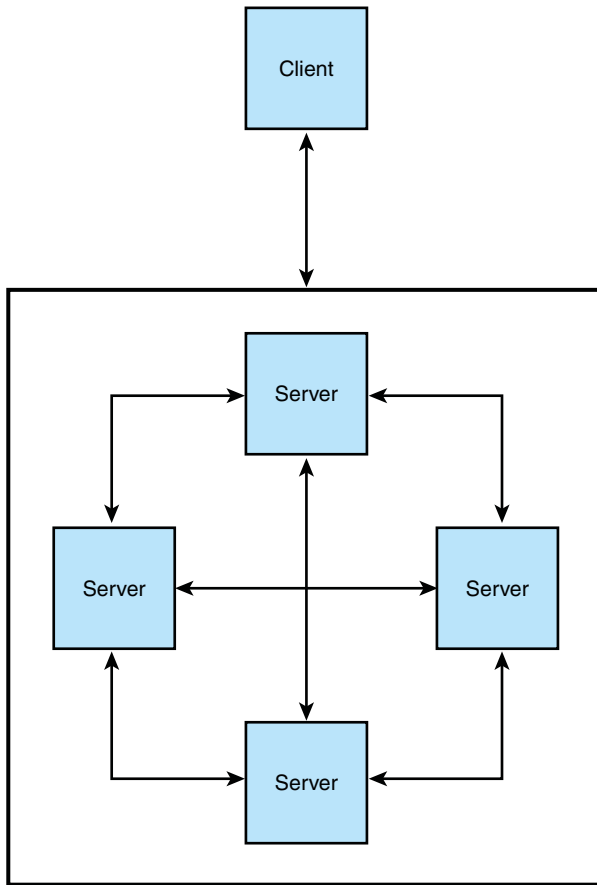


Figure 1.3 The cloud system.

Indeed, it does look similar to the original mainframe architecture; from a purely black-box perspective, it's almost identical. The big difference is how the server actually achieves its processing capabilities by combining the best of both architectures into one.

HTML5 and Local Storage

But really, can't we do better? Isn't there some new technology coming out that will enable us to merge these two separate systems better?

Yes, there is. Client-server interaction mostly involves web applications. The problem with most modern web applications is that they have to ask the server for everything; they can't run any processing locally, and they certainly can't store enough data locally in a usable format.

In comes HTML5 with a **local storage option**, a way to store application-level information within the browser. This new technology enables you to literally distribute every trusted action between client and server. The best non-web example is **Mercurial**, a distributed version control system.

Mercurial doesn't require a central server to push and pull change-sets from, but instead keeps a local copy of your entire repository within the directory you're working on. You can continue to work, check in, revert, update, merge, branch, whatever, even if you're entirely offline. All the client processing happens locally. If you then want to share your changes with others, you can do so either by manually transferring update files, or using a central sync server. Although it does support having one default sync server to push and pull changes to and from, that's not its limit. It can push and pull from as many servers as you'd like without breaking your local repository. These sync servers are simply designed to provide an authenticated distribution point and ensure that the data its providing is updated only by trusted sources.

What this provides is the ability to create a local application on a client's system that synchronizes its database with your central server and then enables you to perform regular tasks on it. Because you only get information from your central server that you have access to, you can open your server up and let any client connect to it, and you can just plop your permissions on top and only let the client see what you want it to see from the database. Unlike the old client-server interaction, your central server's full power is used, and your client's full power is used. Anything requiring massive processing (such as processing graphics) can still run on the server side so that you don't bog down your client's systems, but simple things such as searching through a database and rendering records into visuals can all happen locally on your client!

The biggest question for the new-age application is where to put each bit of functionality. The best answer to this is through trial-and-error. Other than for security, there's no true set of rules on what should be processed where; it's just a matter of seeing what your client is capable of. For example, some clients are capable of running XSLT natively in their browsers, but some don't do it correctly. If you can convince your clients to simply not use unsupported browsers, you can probably offload all of that to them, but you may need to allow your server to run it if you need to support all browsers.

The Dawn of Mobile Devices

Why are we moving away from an all-client infrastructure? Quite simply, it's because of Apple. Smart phones (such as Blackberrys) had previously been thought of for only large businesses, and even then it was usually just for email. There are two different scales of devices: the mobile touch pad and the mobile smart phone. These devices have revolutionized the way you think of the client-server interaction because you can pick up where you left off with one device on another.

Now look at Netflix for a good example. Netflix has recently announced that it will support both the iPhone and the iPad for viewing movies. Additionally, it recently made available Netflix on console gaming systems, and even on some TVs. It also provides specific streaming boxes if you don't want to buy either of these things. The best thing Netflix offers is the ability to pause a movie and resume play on any other device. You can start watching a movie on your way home from the airport, and pick it up again on your TV when you get home!

So what does that mean in the general sense? Quite simply, users want a seamless interaction between their PC and their mobile devices. They don't want to have two totally different systems, just two different interfaces. They want to pick up right where they left off on their desktop when they move to their laptop or iPad. They want to have everything synced automatically for them without

having to copy files. They also want to have offline support so that when they go into one of those pesky areas where there's no AT&T coverage, they can still continue to work, and the system will simply sync the tasks when it can. These are all things that every developer needs to be thinking about when designing their systems.

Threading, Parallel Processing, and Parallel Computing

Whenever dealing with large applications, you often need to split processes into multiple segments to speed things up. Typically this is done by a simple process of **threading**, which is using the language's internal capabilities to process bits of code simultaneously. Threading uses shared memory; you have access to exactly the same information in each thread without having to handle anything else. You have to worry about locking and using semaphores or some other methodology to prevent your application from having problems when accessing the same data. For example, if you have two threads of execution that each increment a single variable by one, they could both read the data at the same time, increment it by one, and then write that back. In this instance, instead of the ending value of the variable being incremented by 2, it would only be incremented by 1 because you didn't properly lock to ensure that only one thread was operating on that bit of data at the same time. This problem of simultaneous write-ability can also be seen in other forms of segmenting code, but in threading, several built-in methods handle locking for you. Threading typically relies on a scheduler built into the language; code is technically not running at the same time, but instead it's given little bits of time to execute in between other threads.

Parallel processing, on the other hand, doesn't have shared memory. Instead of using the language's built-in capabilities of running code in a scheduler, you're actually using the operating system's capability of executing multiple processes, possibly on different CPUs. Because of how this processing is handled, you can't simply share variables, but you can share resources such as files and other

data on the physical machine. Although you don't have the ability to use semaphores anymore for locking, you still have the native filesystem, and you can use file-locks to prevent your code from overlapping writes. Although this is a much better way to separate out bits of code that can run asynchronously, you still need to build one giant supercomputer to process a lot of data because even the most advanced commodity hardware usually tops out at about 12 virtual cores.

The most complex of asynchronous design is parallel computing. Instead of simply splitting your code into different processes and relying on the attached filesystem and other native sources on your local machine, each machine is a large processing unit with a lot of memory to spare. Although you can use the local filesystem as an extension of the RAM for extra storage, you shouldn't rely on it being there in each process, and it can't be used to share data between processes. Because each process could be on an entirely different system, you need to focus on putting any shared data on some other shared datasource. Typically this is a form of database or shared filesystem. This method of design also scales almost infinitely because it's only limited to how many machines you can run at the same time, and how much bandwidth the machines have between them and the shared data sources. Using this methodology, you don't need to build a supercomputer, but you can instead use smaller processing units in a clustered manor to expand almost infinitely. It also enables you to scale and rescale on-the-fly by simply starting and stopping extra servers whenever the demand changes. This is the typical design used behind almost all cloud-based applications.

How Does Cloud-Based Development Differ from Other Application Development?

Unlike server-based application development, cloud-based application development is focused on splitting the two things that every application needs: compute power and data storage. Essentially, this

is the data and compute power to manipulate that data. Data can also be sorted into different levels: shared and nonshared. Nonshared data is like RAM, small, fast, and accessible but can be removed at any point. Although not all cloud computing environments throw away all your data on a local system when you take it down, it's generally a good idea to think of anything on a local system exactly as you would RAM, and assume that after you're done with that specific thread of execution, it will be gone forever. Any data you want to persist or share between processes needs to be stored in some other form, either a database or a data store.

You can't use the local filesystem to send messages between processes when they need to communicate. For example, if you need to fire off new processes or send a message for an existing process to do something, you have to handle that by using a queue, not just making a file somewhere. If your cloud computing service doesn't offer its own queue service (such as Simple Queue Service with Amazon Web Services), you can do the exact same thing with a locking database (such as MySQL or MsSQL).

The most typical thing that's stored in RAM or on local filesystems is session data in Internet applications. Because all cloud applications have some component of the Internet to it (even if you're not making a browser-based application, you're still talking to your application over some Internet protocol), this is an important bit of design to think about. If at all possible, your application should be exactly like the HTTP or HTTPS protocols, completely sessionless. If you do need to store session data, you need to store it in a shared database, not something local because when you're properly scaling your application, you don't know which process the user will access next, nor do you know which server that process is on. You don't even know if the process that first handled that request is even running because you're operating on a constantly scaling system that could be changing, adapting, and even recovering from failures.

Although the cloud can be used to implement legacy code and save you money, in general you probably want to change things

around instead of simply copying existing systems. It's a good idea to use as much as your cloud provider has to offer, which includes using any shared database systems, queuing systems, and computing systems. All cloud providers offer a **compute cloud**, a utility to execute code against data, but most also offer hosted databases and storage. Some cloud providers even provide queuing systems that can help you send messages between processes.

The most important thing to remember when developing a cloud-based application is that *failure is inevitable*. Instead of spending hours trying to figure out why something failed, *just replace it*. It usually takes only about a minute to launch a new server, and if you've built everything properly, there won't be any data loss and little impact, if any, from the outside world. You can achieve this by using the proper patterns and design techniques outlined in Part III, "Projects."

If you have experience developing cluster-based applications, you already have a good start to move to the cloud. The main difference between developing with cluster-based applications and cloud-based applications is using existing systems instead of building your own. For example, you may have already built your own proxy system, but many cloud providers offer their own solutions, which will cost you less and require much less maintenance work.

What to Avoid

As soon as people hear they can put their application "in the cloud," they assume the application is now simply infallible, super scalable, and will easily adapt and save money. *Just because your application is running in the cloud doesn't make it scalable or infallible*. You need to actually build your application around the cloud to take the full advantages it has to offer. Most cloud offerings use commodity hardware in a clustered fashion; it's exactly as likely to fail as any desktop put under the same pressure. *Get ready for failures*. The goal of cloud computing isn't to avoid failures but to be prepared and recover from them.

One of the worst things you can do is make everything rely on one single point of failure, such as a server responsible for maintaining some data or handing out requests. The biggest part of making an application run in the cloud is avoiding bottlenecks. In the past, the biggest bottleneck used to be physical servers, bandwidth, and money. With cloud-computing services, you suddenly are no longer limited by these, so you now can focus on other areas where things could get stuck in a bottleneck. Typically this is something like generating sequential numbers or some other non-threadable process. *Whenever possible, avoid using sequential numbers for IDs.* Instead, try using random UUIDs, which are almost guaranteed to never overlap. Switching away from requiring a single blocking thread to generate the next number in a series for IDs means that you no longer rely on a single point of failure, and you avoid that bottleneck. This process scales almost infinitely because you can simply throw more threads or processes at the service when you have more demand.

Getting Started in the Cloud

The first thing to tackle when working with cloud-based applications is creating your instance image. This is the core set of code that will be used and can either be everything you need or only the base subset of what all your instances need. The more you put into an image, the larger it will be and the slower it will start up, but if you don't put enough onto an instance, it will take even longer to install those extra features before the rest of your services can start. You can use a simple package-management system on a core image to limit what you need to update to a single image, configuring extra packages on boot, or you can build multiple images, each with its own set of packages and update each of them individually. Each cloud-provider enables you to create an image and then set specific instance-level data when running a single instance of that image. You can use this configuration data to load up extra packages, perform updates, or just simply pass in secure information that you would not want to bundle in your image.

In Amazon Web Services, this instance data is typically used to store your AWS credentials so that you can access other services from that instance.

Next, you need to decouple your data from your processing threads. You want to process all data locally and then upload your results to some shared data storage system. For Amazon Web Services, this would be Simple Storage Service (S3) for large data or SimpleDB (SDB) for small searchable data. If you prefer to use a relational database instead of SDB, you can also use MySQL in the cloud.

After you know *where* to store your data, you need to know when to process it and why you are processing it. This is typically done with a **messaging** method. If you've ever dealt with Aspect-Oriented programming languages (such as Objective-C), you know that you already deal with messaging in your programming, but most of it is handled behind the scenes. In a **message queue**, you send a message with your request for a specific processing instruction, such as a function call, to a shared and lockable data source, and other processes listen on that queue for instructions. If a process picks up an instruction it can handle, it locks that message from being read by anyone else, processes the instruction, and then pushes the resulting data to another shared data source, and finally deletes the message so that no other processes will reprocess the same instruction. If your cloud-provider doesn't offer a solution to this, you can simply use a locking database such as MySQL or MSSQL to create your own queue.

When you know how to send messages between processes, you have to manage locking between processes to ensure that multiple processes don't override each other when accessing the same data. If you use MySQL or MsSQL, you can simply use the locking capabilities provided there and lock on tables or on specific rows if possible. Be wary, though, of locking tables because this will hold up other threads, and if something goes wrong with the process that created the lock before it can free it, you'll end up in a dead-lock state. You can handle this in several ways, but most of them are better documented in the manual for your respective database.

Locking in a nonrelational database, however, usually is an inexact science and involves a lot of waiting. Nonrelational databases, such as SDB, also have problems with **eventual consistency**, meaning that even if you write your own lock into a database, you'll have to wait a reasonable amount of time to make sure it wasn't overridden by someone else before you can actually assume you have the lock. Even with waiting, this isn't a guarantee because you can't actually put a time limit on "eventually." Even with the best practices, you can't reliably use any nonconsistent database for locking. Fortunately, SimpleDB now provides the option to use consistent read and write, but then you lose the advantages provided by eventual consistency.

Selecting a Cloud Pattern

After you know what you're building, you need to select the cloud patterns to use to make your application take full advantage of the cloud. This is more of an art than a science, so read over the introduction on each pattern before selecting one. You'll probably need to select multiple patterns to implement your full application, so don't try to find one pattern to fit all your needs. If you find a case study in Part II, "Patterns," that closely matches your situation, you can use that as a good starting point and build from there. You can also jump directly to Part III and find the specific pattern that fits your needs.

In general, you should try to split your application as much as possible and use the patterns as needed. Most applications can benefit from the clustering patterns in Chapter 7, "Clustering," because just about every cloud application will be based on the web patterns shown there. If you work with asynchronous processing of data, see Chapter 6, "Executing Actions on Data." If you need to access data from outside of the cloud, see Chapter 5, "Designing an Architecture." If you're just starting and you need to know how to build an instance and start developing from scratch, start at Chapter 4, "Designing an Image," and continue from there.

Implementing a Cloud Pattern

After you select your patterns, you need to put them into practice. The last section of each pattern includes details and code examples for Amazon Web Services, but this code can easily be extracted and used in almost any cloud-based system. As cloud providers continue to grow, they're also continuing to merge to provide one unified set of offerings. The **boto** python library currently supports Amazon Web Services, Google Storage, and Eucalyptus. Work is also being done to bring in the Rackspace cloud, which uses the free and open source **Open Stack** library. Because all the examples provided in this book use Python and boto, many of these examples can be easily transitioned to any number of cloud providing platforms with little or no code modifications.

This page intentionally left blank

Making Software a Service

Developing your Software as a Service (SaaS) takes you away from the dark ages of programming and into the new age in which copyright protection, DMA, and pirating don't exist. In the current age of computing, people don't expect to pay for software but instead prefer to pay for the support and other services that come with it. When was the last time anyone paid for a web browser? With the advent of Open Source applications, the majority of paid software is moving to hosted systems which rely less on the users' physical machines. This means you don't need to support more hardware and other software that may conflict with your software, for example, permissions, firewalls, and antivirus software.

Instead of developing a simple desktop application that you need to defend and protect against pirating and cloning, you can develop your software as a service; releasing updates and new content seamlessly while charging your users on a monthly basis. With this method, you can charge your customers a small monthly fee instead of making them pay a large amount for the program upfront, and you can make more money in the long run. For example, many people pirate Microsoft Office instead of shelling out \$300 upfront for a legal copy, whereas if it were offered software online in a format such as Google Docs, those same people might gladly pay \$12.50 a month for the service. Not only do they get a web-based version that they can use on any computer, but everything they save is stored online and backed up. After two years of that user paying for your service, you've made as much money

from that client as the desktop version, plus you're ensuring that they'll stay with you as long as they want to have access to those documents. However, if your users use the software for a month and decide they don't like it, they don't need to continue the subscription, and they have lost only a small amount of money. If you offer a trial-based subscription, users can test your software at *no* cost, which means they're more likely to sign up.

Tools Used in This Book

You need to take a look at some of the tools used throughout this book. For the examples, the boto Python library is used to communicate with Amazon Web Services. This library is currently the most full-featured Python library for interacting with AWS, and it's one I helped to develop. It's relatively easy to install and configure, so you can now receive a few brief instructions here. boto currently works only with Python 2.5 to 2.7, not Python 3. It's recommended that you use Python 2.6 for the purposes of this book.

Signing Up for Amazon Web Services

Before installing the libraries required to communicate with Amazon Web Services, you need to sign up for an account and any services you need. This can be done by going to <http://aws.amazon.com/> and choosing Sign Up Now and following the instructions. You need to provide a credit card to bill you for usage, but you won't actually be billed until the end of each month. You can log in here at any time to sign up for more services. You pay for only what you use, so don't worry about accidentally signing up for too many things. At a minimum, you need to sign up for the following services:

- Elastic Compute Cloud (EC2)
- Simple Storage Service (S3)
- SimpleDB
- Simple Queue Service (SQS)

After you create your account, log in to your portal by clicking Account and then choosing Security Credentials. Here you can see your Access Credentials, which will be required in the configuration section later. At any given time you may have two Access keys associated with your account, which are your private credentials to access Amazon Web Services. You may also inactivate any of these keys, which helps when migrating to a new set of credentials because you may have two active until everything is migrated over to your new keys.

Installing boto

You can install boto in several different ways, but the best way to make sure you're using the latest code is to download the source from github at <http://github.com/boto/boto>. There are several different ways to download this code, but the easiest is to just click the Downloads button and choose a version to download.

Although the master branch is typically okay for development purposes, you probably want to just download the latest tag because that's guaranteed to be stable, and all the tests have been run against it before bundling. You need to download that to your local disk and unpack it before continuing.

The next step will be to actually install the boto package. As with any Python package, this is done using the `setup.py` file, with either the `install` or `develop` command. Open up a terminal, or command shell on Windows, change the directory to where you downloaded the boto source code, and run

```
$ python setup.py install
```

Depending on what type of system you run, you may have to do this as root or administrator. On UNIX-based systems, this can be done by prepending `sudo` to the command:

```
$ sudo python setup.py install
```

On Windows, you should be prompted for your administrative login if it's required, although most likely it's not.

Setting Up the Environment

Although there are many ways to set up your environment for boto, use the one that's also compatible with using the downloaded Amazon Tools, which you can find at <http://aws.amazon.com/developertools>. Each service has its own set of command-line-based developer tools written in Java, and most of them enable you to also use the configuration file shown here to set up your credentials. Name this file `credentials.cfg` and put it somewhere easily identified:

```
AWSAccessKeyID=MyAccessKey
AWSSecretKey=MySecretKey
```

You can make this the active credential file by setting an environment variable `AWS_CREDENTIAL_FILE` and pointing it to the full location of this file. On bash-based shells, this can be done with the following:

```
export AWS_CREDENTIAL_FILE=/full/path/to/credentials.cfg
```

You can also add this to your shell's RC file, such as `.bashrc` or `.zshrc`, or add the following to your `.tcshrc` if you use T-Shell instead:

```
setenv AWS_CREDENTIAL_FILE=/full/path/to/credentials.cfg
```

For boto, create a `boto.cfg` that enables you to configure some of the more boto-specific aspects of your systems. Just like in the previous example, you need to make this file and then set an environment variable, this time `BOTO_CONFIG`, to point to the full path of that file. Although this configuration file isn't completely necessary, some things can be useful for debugging purposes, so go ahead and make your `boto.cfg`:

```
# File: boto.cfg
# Imitate some EC2 configs
[Instance]
local-ipv4 = 127.0.0.1
local-hostname = localhost
security-groups = default
public-ipv4 = 127.0.0.1
public-hostname = my-public-hostname.local
```

```
hostname = localhost
instance-type = m1.small
instance-id = i-00000000

# Set the default SDB domain
[DB]
db_name = default

# Set up base logging
[loggers]
keys=root,boto

[handlers]
keys=hand01

[formatters]
keys=form01

[logger_boto]
level=INFO
handlers=hand01

[logger_root]
level=INFO
handlers=hand01

[handler_hand01]
class=StreamHandler
level=INFO
formatter=form01
args=(sys.stdout,)

[formatter_form01]
format=%(asctime)s [%(name)s] %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The first thing to do here is set up an `[Instance]` section that makes your local environment act like an EC2 instance. This section is automatically added when you launch a boto-based EC2 instance by the startup scripts that run there. These configuration

options may be referenced by your scripts later, so adding this section means you can test those locally before launching an EC2 instance.

Next, set the default SimpleDB domain to “default,” which will be used in your Object Relational Mappings you’ll experiment with later in this chapter. For now, all you need to know is that this will store all your examples and tests in a domain called “default,” and that you’ll create this domain in the following testing section.

Finally, you set up a few configuration options for the Python logging module, which specifies that all logging should go to standard output, so you’ll see it when running from a console. These configuration options can be custom configured to output the logging to a file, and any other format you may want, but for the basics here just dump it to your screen and show only log messages above the `INFO` level. If you encounter any issues, you can drop this down to `DEBUG` to see the raw queries being sent to AWS.

Testing It All

If you installed and configured boto as provided in the previous steps, you should be able to launch a Python instance and run the following sequence of commands:

```
>>> import boto
>>> sdb = boto.connect_sdb()
>>> sdb.create_domain("default")
```

The preceding code can test your connectivity to SimpleDB and create the default domain referenced in the previous configuration section. This can be useful in later sections in this chapter, so make sure you don’t get any errors. If you get an error message indicating you haven’t signed up for the service, you need to go to the AWS portal and make sure to sign up for SimpleDB. If you get another error, you may have configured something incorrectly, so just check with that error to see what the problem may have been. If you’re having issues, you can always head over to the boto home page: <http://github.com/boto/boto> or ask for help in the boto users group: <http://groups.google.com/group/boto-users>.

What Does Your Application Need?

After you have the basic requirements for your application and decide what you need to implement, you can then begin to describe what you need to implement this application. Typically this is not a question that you think about when creating smaller scale applications because you have everything you need in a single box. Instead of looking at everything together as one complete unit or “box,” you need to split out what you actually need and identify what cloud services you can use to fit these requirements. Typical applications need the following:

- Compute power
- Fast temporary storage
- Large long-term storage
- Small queryable long-term storage
- Communication between components or modules

Think about this application as a typical nonstatic website that requires some sort of execution environment or web server, such as an e-commerce site or web blog. When a request comes in, you need to return an HTML page, or perhaps an XML or JSON representation of just the data, that may be either static or dynamically created. To determine this, you need to process the actual request using your compute power. This process also requires fast temporary storage to store the request and build the response. It may also require you to pull information about the users out of a queryable long-term storage location. After you look up the users' information, you may need to pull out some larger long-term storage information, such as a picture that they may have requested or a specific blog entry that is too large to store in a smaller queryable storage engine. If the users request to upload a picture, you may have to store that image in your larger long-term storage engine and then request that the image be resized to multiple sizes, so it may be used for a thumbnail image. Each of these requirements your application has on the backend may be solved by using services offered by your cloud provider.

If you expand this simple website to include any service, you can realize that all your applications need the same exact thing. If you split apart this application into multiple layers, you can begin to understand what it truly means to build SaaS, instead of just the typical desktop application. One major advantage of SaaS is that it lends itself to subscription-based software, which doesn't require complex licensing or distribution points, which not only cuts cost, but also ensures that you won't have to worry about pirating. Because you're actually providing a service, you're locking your clients into paying you every time that they want to use the service. Clients also prefer this method because, just like with a cloud-hosting provider, they don't have to pay as much upfront, and they can typically buy in a small trial account to see if it will work for them. They also don't have to invest in any local hardware and can access their information and services from any Internet access. This type of application moves away from the requirements of having big applications on your client's systems to processing everything on your servers, which means clients need less money to get into your application.

Taking a look back at your website, you can see that there are three main layers of this application. This is commonly referred to as a three-tier application pattern and has been used for years to develop SaaS. The three layers include the data layer to store all your long-term needs, the application layer to process your data, and the client or presentation layer to present the data and the processes you can perform for your client.

Data Layer

The data layer is the base of your entire application, storing all the dynamic information for your application. In most applications, this is actually split into two parts. One part is the large, slow storage used to store any file-like objects or any data that is too large to store in a smaller storage system. This is typically provided for

you by a network-attached-storage type of system provided by your cloud hosting solution. In Amazon Web Services, this is called **Simple Storage Service** or **S3**.

Another large part of this layer is the small, fast, and queryable information. In most typical systems, this is handled by a database. This is no different in cloud-based applications, except for how you host this database.

Introducing the AWS Databases

In Amazon Web Services, you actually have two different ways to host this database. One option is a nonrelational database, known as SimpleDB or SDB, which can be confusing initially to grasp but in general is much cheaper to run and scales automatically. This nonrelational database is currently the cheapest and easiest to scale database provided by Amazon Web Services because you don't have to pay anything except for what you actually use. As such, it can be considered a true cloud service, instead of just an adaptation on top of existing cloud services. Additionally, this database scales up to one billion key-value pairs per domain automatically, and you don't have to worry about over-using it because it's built using the same architecture as S3. This database is quite efficient at storing and retrieving data if you build your application to use with it, but if you're looking at doing complex queries, it doesn't handle that well. If you can think of your application in simple terms relating directly to objects, you can most likely use this database. If, however, you need something more complex, you need to use a Relational DB (RDB).

RDB is Amazon's solution for applications that cannot be built using SDB for systems with complex requirements of their databases, such as complex reporting, transactions, or stored procedures. If you need your application to do server-based reports that use complex select queries joining between multiple objects, or you need transactions or stored procedures, you probably need to use

RDB. This new service is Amazon's solution to running your own MySQL database in the cloud and is actually nothing more than an Amazon-managed solution. You can use this solution if you're comfortable with using MySQL because it enables you to have Amazon manage your database for you, so you don't have to worry about any of the IT-level details. It has support for cloning, backing up, and restoring based on snapshots or points-in-time. In the near future, Amazon will be releasing support for more database engines and expanding its solutions to support high availability (write clustering) and read-only clustering.

If you can't figure out which solution you need to use, you can always use both. If you need the flexibility and power of SDB, use that for creating your objects, and then run scripts to push that data to MySQL for reporting purposes. In general, if you can use SDB, you probably should because it is generally a lot easier to use. SDB is split into a simple three-level hierarchy of domain, item, and key-value pairs. A domain is almost identical to a "database" in a typical relational DB; an Item can be thought of as a table that doesn't require any schema, and each item may have multiple key-value pairs below it that can be thought of as the columns and values in each item. Because SDB is schema-less, it doesn't require you to predefine the possible keys that can be under each item, so you can push multiple item types under the same domain. Figure 2.1 illustrates the relation between the three levels.

In Figure 2.1, the connection between item to key-value pairs is a many-to-one relation, so you can have multiple key-value pairs for each item. Additionally, the keys are not unique, so you can have multiple key-value pairs with the same value, which is essentially the same thing as a key having multiple values.

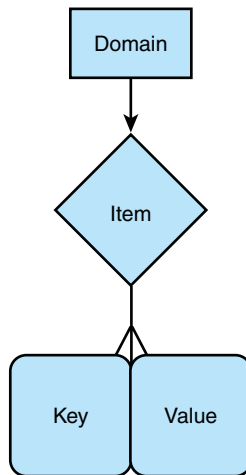


Figure 2.1 The SDB hierarchy.

Connecting to SDB

Connecting to SDB is quite easy using the boto communication library. Assuming you already have your boto configuration environment set up, all you need to do is use the proper connection methods:

```
>>> import boto
>>> sdb = boto.connect_sdb()
>>> db = sdb.get_domain("my_domain_name")
>>> db.get_item("item_name")
```

This returns a single item by its name, which is logically equivalent to selecting all attributes by an ID from a standard database. You can also perform simple queries on the database, as shown here:

```
>>> db.select("SELECT * FROM `my_domain_name` WHERE `name`  
↳LIKE '%foo%' ORDER BY `name` DESC")
```

The preceding example works exactly like a standard relational DB query does, returning all attributes of any item that contains a key `name` that has `foo` in any location of any result, sorting by `name` in descending order. SDB sorts and operates by lexicographical comparison and handles only string values, so it doesn't understand that `[nd]2` is less than `[nd]1`. The SDB documentation provides more details on this query language for more complex requests.

Using an Object Relational Mapping

boto also provides a simple persistence layer to translate all values so that they can be lexicographically sorted and searched for properly. This persistence layer operates much like the DB layer of Django, which it's based on. Designing an object is quite simple; you can read more about it in the boto documentation, but the basics can be seen here:

```
from boto.sdb.db.model import Model
from boto.sdb.db.property import StringProperty, IntegerProperty,
    ReferenceProperty, ListProperty

class SimpleObject(Model):
    """A simple object to show how SDB
    Persistence works in boto"""
    name = StringProperty()
    some_number = IntegerProperty()
    multi_value_property = ListProperty(str)

class AnotherObject(Model):
    """A second SDB object used to show how references work"""
    name = StringProperty()
    object_link = ReferenceProperty(SimpleObject,
        collection_name="other_objects")
```

This code creates two classes (which can be thought of like tables) and a `SimpleObject`, which contains a `name`, `number`, and `multi-valued` property of strings. The `number` is automatically converted by adding the proper value to the value set and properly loaded back by subtracting this number. This conversion ensures that the `number` stored in SDB is always positive, so lexicographical sorting

and comparison always works. The multivalue property acts just like a standard python list, enabling you to store multiple values in it and even removing values. Each time you save the object, everything that was in there is overridden. Each object also has an `id` property by default that is actually the name of the item because that is a unique ID. It uses Python's UUID module to generate this ID automatically if you don't manually set it. This UUID module generates completely random and unique strings, so you don't rely on a single point of failure to generate sequential numbers. The `collection_name` attribute on the `object_link` property of `AnotherObject` is optional but enables you to specify the property name that is automatically created on the `SimpleObject`. This reverse reference is generated for you automatically when you import the second object.

boto enables you to create and query on these objects in the database in another simple manor. It provides a few unique methods that use the values available in the SDB connection objects of boto for you so that you don't have to worry about building your query. To create an object, you can use the following code:

```
>>> my_obj = SimpleObject("object_id")
>>> my_obj.name = "My Object Name"
>>> my_obj.some_number = 1234
>>> my_obj.multi_value_property = ["foo", "bar"]
>>> my_obj.put()
>>> my_second_obj = AnotherObject()
>>> my_second_obj = "Second Object"
>>> my_second_obj.object_link = my_obj
>>> my_second_obj.put()
```

To create the link to the second object, you have to actually save the first object unless you specify the ID manually. If you don't specify an ID, it will be set automatically for you when you call the `put` method. In this example, the ID of the first object is set but not for the second object.

To select an object given an ID, you can use the following code:

```
>>> my_obj = SimpleObject.get_by_id("object_id")
```

This call returns an instance of the object and enables you to retrieve any of the attributes contained in it. There is also a “lazy” reference to the second object, which is not actually fetched until you specifically request it:

```
>>> my_obj.name
u'My Object Name'
>>> my_obj.some_number
1234
>>> my_obj.multi_value_property
[u'foo', u'bar']
>>> my_obj.other_objects.next().name
u'Second Object'
```

You call `next()` on the `other_objects` property because what's returned is actually a `Query` object. This object operates exactly like a generator and only performs the SDB query if you actually iterate over it. Because of this, you can't do something like this:

```
>>> my_obj.other_objects[0]
```

This feature is implemented for performance reasons because the query could actually be a list of thousands of records, and performing a SDB request would consume a lot of unnecessary resources unless you're actually looking for that property. Additionally, because it is a query, you can filter on it just like any other query:

```
>>> query = my_obj.other_objects
>>> query.filter("name like", "%Other")
>>> query.order("-name")
>>> for obj in query:
... 
```

In the preceding code, you would then be looping over each object that has a name ending with `Other`, sorting in descending order on the name. After returning all matching results, a `StopIteration` exception is raised, which results in the loop terminating.

Application Layer

The application layer is where you'll probably spend most of your time because it is the heart and soul of any SaaS system. This is where your code translates data and requests into actions, changing, manipulating, and returning data based on inputs from users, or other systems. This is the only layer that you have to actually maintain and scale, and even then, some cloud providers offer you unique solutions to handle that automatically for you. In Google AppEngine, this is handled automatically for you. In Amazon Web Services, this can be handled semi-automatically for you by using Auto-Scaling Groups, for which you can set rules on when to start and stop instances based on load averages or other metrics.

Your application layer is built on top of a base image that you created and may also contain scripts that tell it to update or add more code to that running instance. It should be designed to be as modular as possible and enable you to launch new modules without impacting the old ones. This layer should be behind a proxy system that hides how many actual modules are in existence. Amazon enables you to do this by providing a simple service known as Elastic Load Balancing, or ELB.

Using Elastic Load Balancing

Amazon's **Elastic Load Balancing**, or **ELB**, can be used simply and cheaply to proxy all requests to your modules based on their Instance ID. **ELB** is even smart enough to proxy only to systems that are actually live and processing, so you don't have to worry about server failures causing long-term service disruptions. **ELB** can be set up to proxy HTTP or standard TCP ports. This is simple to accomplish using code and can even be done on the actual instance as it starts, so it can register itself when it's ready to accept connections. This, combined with **Auto-Scaling Groups**, can quickly and easily scale your applications seamlessly in a matter of minutes without any human interaction. If, however, you want more control over your applications, you can just use **ELB** without **Auto-Scaling Groups** and launch new modules manually.

Creating and managing ELBs is quite easy to accomplish using boto and the `elbadmin` command-line tool that I created, which comes with boto. Detailed usage of this tool can be found by running it on the command line with no arguments:

```
% elbadmin
Usage: elbadmin [options] [command]
Commands:
    list|ls                    List all Elastic Load Balancers
    delete <name>            Delete ELB <name>
    get <name>                Get all instances associated
                             with <name>
    create <name>             Create an ELB
    add <name> <instance>     Add <instance> in ELB <name>
    remove|rm <name> <instance> Remove <instance> from ELB
                             <name>
    enable|en <name> <zone>   Enable Zone <zone> for ELB
                             <name>
    disable <name> <zone>    Disable Zone <zone> for ELB
                             <name>

Options:
    --version                show program's version number and exit
    -h, --help              show this help message and exit
    -z ZONES, --zone=ZONES
                             Operate on zone
    -l LISTENERS, --listener=LISTENERS
                             Specify Listener in,out,proto
```

The first thing to do when starting out is to create a new ELB. This can be done simply as shown here:

```
% elbadmin -l 80,80,http -z us-east-1a create test
Name: test
DNS Name: test-68924542.us-east-1.elb.amazonaws.com
```

```
Listeners
-----
IN      OUT      PROTO
80      80      HTTP
```

Zones

us-east-1a

Instances

You must pass at least one listener and one zone as arguments to create the instance. Each zone takes the same distribution of requests, so if you don't have the same amount of servers in each zone, the requests will be distributed unevenly. For anything other than just standard HTTP, use the `tcp` protocol instead of `http`. Note the `DNS Name` returned by this command, which can also be retrieved by using the `elbadmin get` command. This command can also be used at a later time to retrieve all the zones and instances being proxied to by this specific ELB. The DNS Name can be pointed to by a CNAME in your own domain name. This *must* be a CNAME and not a standard A record because the domain name may point to multiple IP addresses, and those IP addresses may change over time.

Recently, Amazon also released support for adding SSL termination to an ELB by means of the HTTPS protocol. You can find instructions for how to do this on Amazon's web page. At the time of this writing, boto does not support this, so you need to use the command-line tools provided by Amazon to set this up. The most typical use for this will be to proxy port 80 to port 443 using HTTPS. Check with the boto home page for updates on how to do this using the `elbadmin` command-line script.

Adding Servers to the Load Balancer

After you have your ELB created, it's easy to add a new instance to route your incoming requests to. This can be done using the `elbadmin add` command:

```
% elbadmin add test i-2308974
```


This instance must be in an enabled zone for requests to be proxied. You can add instances that are not in an enabled zone, but requests are not proxied until you enable it. This can be used for debugging purposes because you can disable a whole zone of instances if you suspect a problem in that zone. Amazon does offer a service level agreement (SLA), ensuring that it will have 99% availability, but this is not limited to a single zone, thus at any given time, three of the four zones may be down. (Although this has never happened.)

It's generally considered a good idea to use at least two different zones in the event one of them fails. This enables you the greatest flexibility because you can balance out your requests and even take down a single instance at a time without effecting the service. From a developer's perspective, this is the most ideal situation you could ever have because you can literally do upgrades in a matter of minutes without having almost any impact to your customers by upgrading a single server at a time, taking it out of the load balancer while you perform the upgrade.

Although ELB can usually detect and stop proxying requests quickly when an instance fails, it's generally a good idea to remove an instance from the balancer before stopping it. If you're intentionally replacing an instance, you should first verify that the new instance is up and ready, add it to the load balancer, remove the old instance, and then kill it. This can be done with the following three commands provided in boto package:

```
% elbadmin add test i-2308974
% elbadmin rm test i-0983123
% kill_instance i-0983123
```

The last command actually terminates the instance, so be sure there's nothing on there you need to save, such as log files, before running this command. After each of these `elbadmin` commands, the full status of that load balancer is printed, so be sure before running the next command that the previous command succeeded. If a failure is reported, it's most likely because of an invalid instance ID, so be sure you're copying the instance IDs exactly. One useful

tool for this process is the `list_instances` command, also provided in the boto package:

```
% list_instances
```

ID	Zone	Groups	Hostname
i-69c3e401	us-east-1a	Wordpress	.compute-1.amazonaws.com
i-e4675a8c	us-east-1c	default	.compute-1.amazonaws.com
i-e6675a8e	us-east-1d	default	.compute-1.amazonaws.com
i-1a665b72	us-east-1a	default	.compute-1.amazonaws.com

This command prints out the instance IDs, Zone, Security Groups, and public hostname of all instances currently running in your account, sorted ascending by start date. The last instances launched will be at the bottom of this list, so be sure to get the right instance when you're adding the newest one to your ELB. The combination of these powerful yet simple tools makes it easy to manage your instances and ELB by hand.

Although the load balancer is cheap (about 2.5 cents per hour plus bandwidth usage), it's not free. After you finish with your load balancer, remove it with the following command:

```
% elbadmin delete test
```

Automatically Registering an Instance with a Load Balancer

If you use a boto pyami instance, you can easily tell when an instance is finished loading by checking for the email sent to the address you specify in the `Notification` section of the configuration metadata passed in at startup. An example of a configuration section using gmail as the smtp server is shown here:

```
[Notification]
smtp_host = smtp.gmail.com
smtp_port = 587
```

```

smtp_tls = True
smtp_user = my-sending-user@gmail.com
smtp_pass = MY_PASSWORD
smtp_from = my-sending-user@gmail.com
smtp_to = my-recipient@gmail.com

```

Assuming there were no error messages, your instance should be up and fully functional. If you want the instance to automatically register itself when it's finished loading, add an installer to your queue at the end of your other installers. Ensure that this is done after all your other installers finish so that you add only the instance if it's safe. A simple installer can be created like the one here for ubuntu:

```

from boto.pyami.installers.ubuntu.installer import Installer
import boto

class ELBRegister(Installer):
    """Register this instance with a specific ELB"""
    def install(self):
        """Register with the ELB"""
        # code here to verify that you're
        # successfully installed and running
        elb_name = boto.config.get("ELB", "name")
        elb = boto.connect_elb()
        b = elb.get_all_load_balancers(elb_name)
        if len(b) < 1:
            raise Exception, "No Load balancer found"
        b = b[0]
        b.register_instances([boto.config.get_instance
            ➤("instance_id")])
    def main(self):
        self.install()

```

This requires you to set your configuration file on boot to contain a section called `ELB` with one value `name` that contains the name of the balancer to register to. You could also easily adapt this installer to use multiple balancers if that's what you need. Although this installer will be called only if all the other installers before it succeed, it's still a good idea to test anything important before actually registering yourself with your balancer.

HTTP and REST

Now that you have your instances proxied and ready to accept requests, it's time to think about how to accept requests. In general, it's a bad practice to reinvent the wheel when you can just use another protocol that's already been established and well tested. There are entire books on using HTTP and REST to build your own SaaS, but this section provides the basic details.

Although you can use HTTP in many ways, including SOAP, the simplest of all these is Representational State Transfer (REST), which was officially defined in 2000 by Roy Fielding in a doctoral dissertation “Architectural Styles and the Design of Network-based Software Architectures” (<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>). It uses HTTP as a communication medium and is designed around the fundamental idea that HTTP already defines how to handle method names, authentication, and many other things needed when working with these types of communications. HTTP is split into two different sections: the header and the body (not to be confused with the HTML `<head>` and `<body>` tags), each of which is fully used by REST.

This book uses REST and XML for most of the examples, but this is not the only option and may not even suite your specific needs. For example, SOAP is still quite popular for many people because of how well it integrates with Java. It also makes it easy for other developers to integrate with your APIs if you provide them with a Web Service Definition Language (WSDL) that describes exactly how a system should use your API. The important point here is that the HTTP protocol is highly supported across systems and is one of the easiest to use in many applications because much of the lower-level details, such as authentication, are already taken care of.

The Header

The HTTP header describes exactly who the message is designed for, and what method the user is instantiating on the recipient end. REST uses this header for multiple purposes. HTTP method

names can be used to define the method called and the arguments (path) which is sent to that method. The HTTP header also includes a name, which can be used to differentiate between applications running on the same port. This shouldn't be used for anything other than differentiating between applications because it's actually the DNS name and shouldn't be used for anything other than a representation of the server's address.

The method name and path are both passed into the application. Typically you want to use the path to define the module, package, or object to use to call your function. The method name is typically used to determine what function to call on that module, package, or object. Lastly, the path also contains additional arguments after the question mark (?) that usually are passed in as arguments to your function. Now take a look at a typical HTTP request:

```
GET /module_name/id_argument?param1=value1&param2=value2
```

In this example, most applications would call `module_name.`

```
get(id_argument,param1=value1,param2=value2) or  
module_name.get(id_argument,{param1=value1,param2=value2}).
```

By using this simple mapping mechanism, you're decoupling your interface (the web API) from your actual code, and you won't actually need to call your methods from the website. This helps greatly when writing unit tests.

Many libraries out there can handle mapping a URI path to this code, so you should try to find something that matches your needs instead of creating your own. Although REST and RESTful interfaces are defined as using only four methods, most proxies and other systems, including every modern web browser, support adding your own custom methods. Although many REST developers may frown on it, in general it does work, and when a simple CRUD interface isn't enough, it's much better than overloading an existing function to suit multiple needs. The following sections reference some of the most common HTTP headers and how you can use them in a REST API.

If-Match

The most interesting header that you can provide for is the If-Match header. This header can be used on any method to indicate that the request should be performed only if the conditions in the header represent the current object. This header can be exceptionally useful when you operate with databases that are eventually consistent, but in general, because your requests can be made in rapid succession, it's a good idea to allow for this so that they don't overwrite each other. One possible solution to this is to provide for a version number or memento on each object or resource that can then be used to ensure that the user knows what the value was before it replaces it.

In some situations, it may be good to require this field and not accept the special `*` case for anyone other than an administrative user. If you require this field to be sent and you receive a request that doesn't have it, you should respond with an error code of `412` (`Precondition Failed`) and give the users all they need to know to fill in this header properly. If the conditions in this header do not match, you *must* send back a `412` (`Precondition Failed`) response. This header is typically most used when performing `PUT` operations because those operations override what's in the database with new values, and you don't know if someone else may have already overridden what you thought was there.

If-Modified-Since

The If-Modified-Since header is exceptionally useful when you want the client to contain copies of the data so that they can query locally. In general, this is part of a caching system used by most browsers or other clients to ensure that you don't have to send back all the data if it hasn't been changed. The If-Modified-Since header takes an HTTP-date, which must be in GMT, and should return a `304` (`Not Modified`) response with no content.

If-Unmodified-Since

If you don't have an easy way to generate a memento or version ID for your objects, you can also allow for an If-Unmodified-Since header. This header takes a simple HTTP date, formatted in GMT, which is the date the resource was last retrieved by the client. This puts a lot of trust in the client, however, to indicate the proper date. It's generally best to use the If-Modified header instead, unless you have no other choice.

Accept

The Accept header is perhaps the most underestimated header in the entire arsenal. It can be used not only to handle what type of response to give (JSON, XML, and so on), but also to handle what API version you're dealing with. If you need to support multiple versions of your API, you can support this by attaching it to the content type. This can be done by extending the standard content types to include the API version number:

```
Accept: text/xml+app-1.0
```

This enables you to specify not only a revision number (in this case, 1.0) and content type, but also the name of the application so that you can ensure the request came from a client that knew who it was talking to. Traditionally, this header will be used to send either HTML, XML, JSON, or some other format representing the resource or collection being returned.

Authorization

The Authorization header can be used just like a standard HTTP authentication request, encoding both the password and the username in a base64 encoded string, or it can optionally be used to pass in an authentication token that eventually expires. Authentication types vary greatly, so it's up to you to pick the right version for your application. The easiest method is by using the basic HTTP authentication, but then you are sending the username and

password in every single request, so you *must* ensure that you're using SSL if you're concerned about security.

In contrast, if you choose to use a token or signing-based authentication method, the user has to sign the request based on some predetermined key shared between the client and server. In this event, you can hash the entire request in a short string that validates that the request did indeed come from the client. You also need to make sure to send the username or some other unique identifier in this header, but because it's not sending a reversible hash of the password, it's *relatively* safe to send over standard HTTP. We won't go into too much depth here about methods of hashing because there are a wide variety of hashing methods all well-documented online.

The Body

The body of any REST call is typically either XML or JSON. In some situations, it's also possible to send both, depending on the Accept header. This process is fairly well documented and can be used to not only define what type of response to return, but also what version of the protocol the client is using. The body of any request to the system should be in the same format as the response body.

In my applications, I typically use XML only because there are some powerful tools, such as XSLT, that can be used as middleware for authorization purposes. Many clients, however, like the idea of using JSON because most languages serialize and deserialize this quite well. REST doesn't specifically require one form of representation over the other and even enables for the clients to choose which type they want, so this is up to you as the application developer to decide what to support.

Methods

REST has two distinct, important definitions that you need to understand before continuing. A **collection** is a group of objects; in your case this usually is synonymous with either a class in object terms, or a table in database terms. A **resource** is a specific instantiation of a collection, which can be thought of as an instance in object terms or a row in a table in database terms. This book also uses the term **property** to define a single property or attribute on an instance in object terms, or a cell in database terms. Although you can indeed create your own methods, in general you can probably fit most of your needs into one of the method calls listed next.

GET

The GET method is the center of all requests for information. Just like a standard webpage, applications can use the URL in two parts; everything before the first question mark (?) is used as the resource to access, and everything after that is used as query parameters on that resource. The URL pattern can be seen here:

```
/collection_name/resource_id/property_name?query
```

The `resource_id` `property_name` and `query` in the preceding example are all optional, and the query can be applied to any level of the tree. Additionally, this tree could expand exponentially downward if the property is considered a reference. Now take a simple example of a request on a web blog to get all the comments of a post specified by `POST-ID` submitted in 2010. This query could look like this:

```
/posts/POST-ID/comments?submitted=2010%
```

The preceding example queries for the `posts` collection for a specific post identified as `POST-ID`. It then asks for just the property named `comments` and filters specifically for items with the property `submitted` that matches `2010%`.

Responses to this method call can result in a redirection if the resource is actually located at a different path. This can be achieved by sending a proper redirect response code and a `Location` header that points to the actual resource.

A GET on the root of the collection should return a list of all resources under that path. It can also have the optional `?query` on the end to limit these results. It's also a good idea to implement some sort of paging system so that you can return all the results instead of having to limit because of HTTP timeouts. In general, it's never a good idea to have a request that takes longer than a few seconds to return on average because most clients will assume this is an error. In general, most proxy systems will time out any connection after a few minutes, so if your result takes longer than a minute, it's time to implement paging.

If you use XML as your communication medium, think about implementing some sort of ATOM style `next` links. These simple tags give you a cursor to the next page of results, so you can store a memento or token of your query and allow your application to pick up where it left off. This token can then be passed in via a query parameter to the same URL that was used in the original request. In general, your `next` link should be the *full* URL to the next page of results. By doing this, you leave yourself open to the largest range of possibilities for implementing your paging system, including having the ability to perform caching on the next page of results, so you can actually start building it before the client even asks for it.

If you use Amazon Web Services and SimpleDB, it's generally a good idea to use the `next_token` provided by a SimpleDB query as your memento. You also need to provide enough information in the next link to build the entire original query, so just using the URL that was originally passed and adding the `next_token` to the end of the query is generally a good idea. Of course if this is a continuation, you have to replace the original `next_token` with the new one.

Performing a GET operation on the root path (`/`) should return a simple index that states all the available resource types and the URLs to those resource types. This machine-readable code should be simple enough that documentation is not required for new developers to build a client to your system. This methodology enables you to change around the URLs of the base resources without modifying your client code, and it enables you build a highly adaptable client that may adapt to new resources easily without making any modifications.

PUT

The HTTP definition of a PUT is *replace*, so if you use this on the base URL of a resource collection, you're actually requesting that everything you don't pass in is deleted, anything new is created, and any existing resources passed in are modified. Because PUT is logically equivalent to a SET operation, this is typically not allowed on a collection.

A PUT on a resource should update or create the record with a specific ID. The section after the collection name is considered the ID of the resource, and if a GET is performed after the PUT, it should return the resource that was just created or updated. PUT is intended as an entire replacement, but in general, if you don't pass in an attribute, that is assumed to be "don't change," whereas if you pass in an empty value for the attribute, you are actually requesting it to be removed or set to blank.

A PUT on a property should change just that specific property for the specified resource. This can be incredibly useful when you're putting files to a resource because those typically won't serialize very well into XML without lots of base64 conversions.

Most PUT requests should return either a 201 `Created` with the object that was just created, which may have been modified due to server application logic, or a 204 `No Content` if there are no changes to the original object request. If you operate with a database that has eventual consistency, it may also be a good idea to instead return a 202 `Accepted` request to indicate that the client should try to fetch the resource at a later time. You should also return an estimate of how long it will be before this object is created.

POST

A POST operation on a collection is defined as a *creation request*. This is the user requesting to add a new resource to the collection without specifying a specific ID. In general, you want to either return a redirection code and a Location header, or at the least the ID of the object you just created (if not the whole object serialized in your representation).

A POST operation on a resource should actually create a sub-object of that resource; although, this is often not used. Traditionally,

browsers don't handle changing form actions to PUT, so a POST is typically treated as a special form of a PUT that takes form-encoded values.

A POST operation on a property is typically used only for uploading files but could also be used as appending a value to a list. In general, this could be considered an append operation, but if your property is a single value, it's probably safe to assume that the client wanted this to be a PUT, not a POST.

DELETE

A DELETE operation on a collection is used to drop the entire collection, so you probably don't want to allow this. If you do allow it, this request should be treated as a request to remove every single resource in the collection.

A DELETE operation on a specific resource is simply a request to remove that specific resource. If there are errors in this request, the client should be presented with a message explaining why the request failed. The resulting error code should explain if the request can be issued again, or if the user is required to perform another operation before reissuing the request. The most typical error message back from this request is a 409 `Conflict`, which indicates that another resource is referencing this resource, and the server is refusing to cascade the delete request. A DELETE may also return a 202 `Accepted` response if the database has eventual consistency.

A DELETE operation on a specific property is identical to a PUT with an empty value for that property. It can be used to delete just a single property from a resource instead of having to send a PUT request. This can also be used as a differentiation between setting a value to blank and removing the value entirely. In programming terms, this the difference between an empty string and `None` or `Null`.

HEAD

A HEAD request on any URL should return the exact same headers as a standard GET request but shouldn't send back the body. This is typically used to get a count of the number of results in a response without retrieving the actual results. In my applications, I use this

to send an additional header `X-Results`, which contains the number of results that would have been retrieved.

OPTIONS

An `OPTIONS` request on any URL returns the methods allowed to be performed on this URL. This can return just the headers with the additional `Accept` header, but it can also return a serialized version of them in the body of the results that describes what each method actually does. This response should be customized for the specific user that made the request, so if the user is not allowed to perform `DELETE` operations on the given resource for any reason, that option should not be returned. This allows the client to specifically hide options that the users aren't allowed to perform so that they don't get error responses.

Authorization Layer

The authorization layer sits just above your application layer but is still on the sever. In most application systems, this is actually integrated directly with your application layer. Although this is generally accepted, it doesn't provide for as much flexibility, and it's a lot harder to code application logic and authorization in the same location. Additionally, if you go to change your authentication, you now have to worry about breaking your application layer and your authentication layer. If you build this as a separate layer, you can most likely pull all authorization directly out of a database, so you don't have to worry about changing your code just for a minor change in business logic.

If you use XML for your object representation, you can use XSLT for your authorization layer, and use some custom tags to pull out this authorization logic directly from your database. If you use an application framework such as Django or Ruby Rails, chances are you already have this layer built for you, either directly or in a third-party module. Check your specific language for how to build your own extensions for XSLT. When you can build your own extensions into your XSLT processor, not only can your filters

be retrieved from a shared filesystem so that you can update them without redistributing them to each of your servers, but you can also pull the exact details about authorization from there. These XSLT filters can be used to specifically hide elements of the response XML that the user shouldn't see. The following example code assumes you've already built a function called `hasAuth` that takes three arguments, authorization type (read, write, delete), object type, and property name:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:app="url/to/app">
  <!-- By default pass through all XML elements -->
  <xsl:template match="@*|node()" priority="-10">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>

  <!-- Object-level permissions -->
  <xsl:template match="/node()">
    <xsl:if test="app:hasAuth('read', current())">
      <xsl:copy>
        <xsl:apply-templates select="@*|node()" mode="property"/>
      </xsl:copy>
    </xsl:if>
  </xsl:template>

  <!-- Property-level permissions -->
  <xsl:template match="node()" mode="property">
    <xsl:if test="app:hasAuth('read', .., current())">
      <xsl:copy>
        <xsl:apply-templates select="@*|node()" />
      </xsl:copy>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

The preceding example is for output from your server, but you could easily adapt this to any method you need by changing the

first argument to each `hasAuth` call to whatever method this is filtering on. You could also easily use this as a base template and pass in the method name to the filter. This example assumes you have an input XML that looks something like the following example:

```
<User id="DEAD-BEAF">
  <name>Foo</name>
  <username>foo</username>
  <email>someone@example.com</email>
</User>
```

Using this example as a base, you could also build filters to operate with JSON or any other representation, but XSLT still seems to be the simplest because you can also use it to create more complex authorization systems, including a complicated group-based authentication, and it can be used to include filters within other filters. If you do need to support sending and receiving information in other formats, you can always use another filter layer on top of your application to translate between them.

Client Layer

After you build your complex application server, you need to focus on making that simple WebAPI usable to your clients. Although developers may be fine with talking XML or JSON to your service, the average user probably won't be, so you need to build a client layer. The biggest concept to understand about this layer is that *you cannot trust anything it sends*. To do so means you're authenticating the client, and no matter what kind of protection you try to do, it's impossible to ensure that the information you get from any client is *your* client.

You must assume that everything you send to your client is viewable by the user. Don't assume the client will hide things that the user shouldn't see. Almost every web service-based vulnerability comes from just blindly trusting data coming from a client, which can often be formed specifically for the purpose of making your application do something you didn't intend. This is the entire basis behind the SQL injection issues many websites still suffer

from. Because of these types of security concerns, you have to ensure that all authentication and authorization happens *outside* of this layer.

You can develop this layer in several different ways, the easiest of which is to expose the API and allow other third-party companies to build their own clients. This is the way that companies such as Twitter handled creating its clients. You can also make your own clients, but by having a well-documented and public-facing API, you expose yourself to having other people and companies developing their own clients. In general, you have to be ready for this event, so it's always a good idea to ensure that the client layer has only the access you want it to have. You can never tell for sure that the client you're talking to is one that you've developed.

Browser-Based Clients

With the release of the HTML5 specifications and local database storage, it's become increasingly easier to make rich HTML and JavaScript-based clients. Although not all browsers support HTML5, and even less support local databases, it's still possible to create a rich experience for the user with just a browser.

Although some browsers do support cross-site Ajax requests, it's generally not a good idea to rely on that. Instead, use something such as Apache or NginX to proxy a sub-URL that can be used by your JavaScript client. Don't be afraid to use Ajax requests for loading data. It's also a good idea to build up a good base set of functions or use an existing library to handle all your serializing and deserializing of your objects, and making Ajax requests and handling caching of objects. You don't want to hit your server more often than absolutely required because HTTP requests are relatively expensive. So it's a good idea to keep a local cache if your browser supports it, but don't forget to either keep the memento provided, or the time when it was retrieved so that you can send an If-Modified-Since header with each request.

If you use a JavaScript-based client layer, it's strongly recommended that you use something such as jQuery to handle all your

cross-browser issues. Even if you hate using JavaScript, jQuery can make creating this client layer simple. This, however, can be tricky because it relies a lot on the browsers feature support, which can be entirely out of your own control. Additionally, this generally doesn't work well for mobile devices because those browsers have much less JavaScript support, and they typically don't have much memory at all.

If you don't want to use JavaScript, Flash is another option. Flash has several libraries for communicating over HTTP using XML, so you should use something that already exists there if you want to support this. If you don't like the representations that these libraries supply, you can also use the `Accept` header and a different version number to allow your system to support multiple different representations. Flash has the advantage of being easy to code in, in addition to providing a much more unified interface. It's also quite heavy and can be easy to forget that this is an untrusted layer. Most mobile devices don't support Flash, so if that's important to you, you need to stick to JavaScript and HTML.

Another option if you don't want to use either HTML/JavaScript or Flash is using a Java Applet. Although there's a lot of possibilities that you can do with these little applets, they're also heavy weight, and if you're trying to get it to work on mobile browsers, this just won't work. Additionally, few mobile devices support either Flash or Java Applets, so if you're going for the most support, you probably want to stick with HTML and JavaScript. Still, if you need more features than Flash or HTML can support, this might be a good option for you.

Native Applications

If you've ever used Twitter, chances are you know that not only is there a web interface, but there are also dozens of native client applications that work on a wide variety of platforms. In your system, you can make as many different client applications as you want to, so eventually you may want to branch out and make some native applications for your users. If you're building that wonderful

new electronic filing cabinet to organize your entire world online, it might be a good idea to also provide some native apps, such as an iPhone application that also communicates to the same API as every other client to bring you the same content wherever you are in the world.

Even if you're not the one building native applications, it may still be a good idea to let other third parties have this option. If you make your API public and let third-party companies make money from creating client layers, you not only provide businesses with a reason to buy into your app, but you also don't need to do any of the development work yourself to increase your growth. Although this may not seem intuitive because people are making a profit off your system, every penny they earn is also bringing you business, and chances are they're marketing as well. Not only does this give your users more options for clients, but it also gives your system more exposure and potential clients.

Summary

Just like cloud providers give you *hardware as a service*, developing your applications as *Software as a Service* gives you another level of benefit, expanding your applications beyond the standard services. By expanding your application to be a service instead of just software, you're giving yourself a huge advantage over any competition. Services are always growing and have infinite potential to keep customers. SaaS gives your customers the same advantage that the cloud gives to you, low initial cost and a reason to keep paying. Everything from business-level applications to the newest games are being transformed from standard single-person applications into services. Don't let your development time go to waste by developing something that will be out of date by the time it's released.

This page intentionally left blank

Index

NUMBERS

409 Conflict error message, 61

412 (Precondition Failed) error code, 55

A

Accept headers, 56

Access Credentials (Amazon Web Services), viewing, 35

adapters

description of, 140

example of, 141-145

Gui-Over-Database application
framework, 141

implementing, 140-141

ORM layers, 141

parts of, 139

usefulness of, 146

using, reasons for, 140

Amazon Web Services, 69

Access Credentials, viewing, 35

account setup, 34

CloudFront

accessing, 78

deleting HTTP distributions, 80

delivering updates via, 78

disabling HTTP
distributions, 79

- disadvantages of, 78
- hierarchy of, 77
- invalidating file caches, 79
- obtaining usage statistics, 78
- private data distributions, 78
- resuming HTTP
 - distributions, 79
- RTMP streams, 77
- serving files from HTTP
 - distributions, 79
- static HTTP distributions, 79
- streaming HTTP
 - distributions, 79
- EBS, 88-91, 129
- EC2, 83
 - creating connection objects, 84
 - creating security groups, 84
 - creating SSH key-pairs, 85
 - filtering images, 86
 - finding images, 86
 - instances, 87-92
 - opening ports to security groups, 85
 - removing rules from security groups, 85
 - reservation objects, 87
 - running images, 87
 - searching for images, 86
 - viewing regions, 84
- ELB, 47, 91-93, 155-158, 279
- legacy patterns, 25
- logging into, 35
- RDS, 95-102
- regions, 70, 84
- Relational Data Service, 231
- S3, 41
 - accessing files, 75
 - billing for usage, 72
 - buckets, 71-73, 77
 - getting file contents as strings, 77
 - hierarchy of, 71
 - keys, 74-77
 - making files public, 76
 - RSS, 73
 - sending files to S3, 75
 - signed URLs, 76
 - SLA, 72
 - using file objects directly, 77
- SDB commands, 175
- SimpleDB, 43, 93-95
- SNS, 102-105, 189
- SQS, 165-166
 - asynchronous procedure calls, 80
 - commands, 175
 - counting messages in queues, 82
 - creating messages in queues, 81-82
 - creating queues, 80
 - default queue timeouts, 81
 - deleting in queues, 83
 - deleting messages from queues, 82
 - finding queues, 81
 - hiding messages in queues, 82
 - launching, 80
 - reading messages from queues, 82
 - sq3 read loops, 165

VPC, 106-108

zones, 70

Amazon.com

clouds and, 6, 22

Content Distribution Network, 77

SimpleDB, 31

Analytics, obtaining via CloudFront, 78

Apache, installing (blog building example), 276-277

Apache2, 254

AppEngine, 108-110

application layers

ELB, Auto-Scaling Groups, 47

n-tier web clusters, 198

applications. See also blogs, building

configuring, 287

CRUD applications,
configuring, 287

deployment strategies, 272. *See also*
blogs, building

development of

cloud-based versus server-based,
11-13

failure and, 13

implementing cloud patterns, 17

instance images, 14

locking processes, 15

messaging and, 15

processing data locally, 15

selecting cloud patterns, 16

sequential numbers as ID, 14

uploading data to shared storage
systems, 15

UUID, 14

Gui-Over-Database application
framework, 141

Installing, blog building example,
275-276

logic (blog building example), 240

Comment handlers, 249-250

DB handlers, 243-247

post handlers, 248

spam classification, 250-253

User handlers, 248

WSGI, 241-250

REST, accepting requests via

collections, 58

DELETE method, 61

GET method, 58-59

HEAD method, 61

headers, 53-56

OPTIONS method, 62

POST method, 60

properties, 58

PUT method, 60

resources, 58

writing REST bodies, 57

SaaS applications, developing

accepting requests via REST,
53-62

application layer, 47-52

authorization layer, 62-64

browser-based clients, 65-66

client layer, 64-67

data layer, 40-46

determining application
requirements, 39-40

HTTP headers, 55-56

launching python instances, 38

- native applications, 66–67
 - REST collections, 58
 - REST method, 58
 - REST methods, 58–62
 - REST properties, 58
 - REST resources, 58
 - setting up EC2 instances, 37
 - setting up SimpleDB
 - domains, 38
 - setting up working
 - environment, 36–38
 - testing SimpleDB
 - connectivity, 38
 - writing REST bodies, 57
 - WSDL, 53
 - scalability, 229
 - self-healing applications, 28
 - architectures**
 - adapters
 - description of, 140
 - example of, 141–145
 - Gui-Over-Database application
 - framework, 141
 - implementing, 140–141
 - ORM layers, 141
 - parts of, 139
 - reasons for using, 140
 - usefulness of, 146
 - balancers
 - ELB, 155–158
 - proxy balancers, 153–158
 - usefulness of, 158
 - Facades, 138
 - description of, 148
 - example of, 149–152
 - implementing, 148–149
 - mappers, 148–152
 - reasons for using, 148
 - request handling, 147
 - usefulness of, 152
 - proxies
 - ELB, 155–158
 - proxy balancers, 137, 153–158
 - reasons for using, 153
 - usefulness of, 158
 - asynchronous procedure calls, SQS, 80**
 - asynchronous requests, 137, 161**
 - ATOM feeds**
 - current page links, 182
 - iterators, 182
 - next page links, 182
 - paging in, 183–185
 - authentication, Twitter, 141**
 - Authorization headers, 56**
 - authorization layer (applications), 62–64**
 - Auto-Scaling Groups, 47**
 - Availability, cloud services and, 30**
-
- ## B
-
- balancers**
 - ELB, 155–158
 - proxy balancers, 153
 - description of, 154
 - example of, 155–158
 - implementing, 154–155
 - usefulness of, 158
 - usefulness of, 158
 - base instances, starting (application deployment strategies), 272–273**

Bayesian filtering systems, spam classification, 250-253

blogs, building, 229

- application logic, 240
 - Comment handlers, 249-250
 - DB handlers, 243-247
 - post handlers, 248
 - spam classification, 250-253
 - User handlers, 248
 - WSGI, 241-250
- deployment strategies
 - bundling images, 278
 - creating proxies, 279
 - installing Apache, 276-277
 - installing applications, 275-276
 - installing software, 273-275
 - starting base instances, 272-273

- Marajo blog building example
 - configuring applications, 287
 - creating handlers, 286
 - creating resources, 284-286
 - creating templates, 288-289
 - custom templates, 289-296
 - initializing web environment, 282-283
 - running applications, 289

presentation, 253

- adding comments, 270-271
- comments, 266-269
- creating posts, 259-261
- deleting posts, 262-263
- editing posts, 263-266
- HTTP Proxy configurations, 254-255

listing posts, 255-259

marking comments as spam/ham, 271-272

viewing comments, 269-270

storage, 229

- Comment objects, 237-240
- database schema, 230
- domains, creating, 231
- Post objects, 230, 234-237
- Relational Data Service, 231
- User objects, 230-234

boto python library, 17

- boto.cfg files, creating, 36
- CloudFront, accessing, 78
- downloading, 35
- EBS, accessing, 89
- EC2, accessing, 84
- ELB, accessing, 92
- installing, 35
- objects
 - creating, 45
 - querying, 46
- pyami images, 133
- RDS, accessing, 96
- requirements for, 34
- SimpleDB, accessing, 94
- SNS, accessing, 103
- software installations, application deployment strategies, 273-275
- VPC, accessing, 106

browsers

- browser-based clients, 65-66
- thin clients as, 3

buckets (S3), 71

- accessing, 72
- deleting, 77
- keys
 - deleting, 77
 - fetching contents of, 76
 - listing, 75-76
 - naming, 74
- naming, 73
- sharing, 72
- uniqueness of, 73

bundling images, blog building**example, 278****bwclient-js, 205**

C

CAN-SPAM Unsubscribe policy, SNS, 189**CAP theorem, eventual consistency, 31****client layer (applications), 64**

- browser-based clients, 65-66
- n-tier web clusters, 197
- native applications, 66-67

cloud providers. See also clouds

- Amazon Web Services, 69
 - CloudFront, 77-80
 - EBS, 88-91
 - EC2, 83-88
 - ELB, 91-93
 - RDS, 95-102
 - regions, 70, 84
 - S3, 71-77
 - SimpleDB, 93-95
 - SNS, 102-105
 - SQS, 80-83

VPC, 106-108

zones, 70

commodity hardware, 25-28

failure, 27-28

Google cloud, 69

performance, complaints
against, 25

Rackspace cloud, 69

scalability, 26

CloudFiles, 112-113**CloudFront**

- accessing, 78
- disadvantages of, 78
- hierarchy of, 77
- HTTP distributions
 - deleting, 80
 - disabling, 79
 - invalidating file caches, 79
 - resuming, 79
 - serving files, 79
 - static distributions, 79
 - streaming distributions, 79
- private data distributions, 78
- RTMP streams, 77
- updates, delivering via, 78
- usage statistics, obtaining, 78

clouds. See also cloud providers

- advantages of, 1-2
- application development,
 - server-based versus cloud-based, 11-13
- cloud services
 - availability and, 30
 - compute services, 24
 - connectivity services, 24

- consistent systems and, 29–30
 - defining, 23
 - legacy patterns, 25–26
 - partition tolerance and, 30–31
 - storage, 23–24
- compute clouds, 13
- defining, 1
- evolution of
- Amazon.com, 6
 - clustering, 5
 - HTML5, 8–9
 - Internet speeds, 5–6
 - local storage, 8–9
 - mainframes, 3
 - mobile devices, 9–10
 - multitasking's role in, 4
 - parallel computing, 11
 - parallel processing, 10–11
 - PC development, 3–4
 - processing speeds, 5–6
 - threading, 10
- migrating to, 26
- origins of, 21–22
- scalability, 26
- CloudServers, 113**
- CloudSites, 113–114**
- clusters, 193**
- cloud evolution, 5
 - graphics and, 5–6
 - hadoop clusters, 174
 - locking clusters, semaphores and
 - description of, 212
 - example of, 213–218
 - implementing, 212
 - parts of, 211
 - reasons for using, 211
 - usefulness of, 218
 - map/reduce clusters
 - description of, 220
 - example of, 222–226
 - implementing, 220
 - parts of, 219
 - reasons for using, 220
 - usefulness of, 226
 - n-tier web clusters
 - application layers, 198
 - client layers, 197
 - database layers, 198
 - description of, 196
 - example of, 198–210
 - filter layers, 198
 - implementing, 197–198
 - parts of, 195–196
 - reasons for using, 196
 - representation layers, 198
 - usefulness of, 210
- collections (objects), 58**
- commands**
- description of, 174
 - example of, 175–179
 - implementing, 174
 - usefulness of, 179
 - using, reasons for, 173
- Comment handlers, building blog application logic, 249–250**
- Comment objects, data storage, 237–240**

comments, building blogs, 266-268

- adding, 270-271

- marking as spam/ham, 271-272

- viewing, 269-270

commodity hardware, 25**compute clouds, 13****compute services, 24****configuring applications, 287****connectivity services, 24****consistent systems, cloud services and, 29-30****Content Distribution Network, 77****CRUD applications, configuring, 287****current page links (ATOM feeds), 182****custom templates, Marajo blog building example**

- edit templates, 292-296

- list templates, 289-292

D

data layer (applications), SimpleDB connections, 40-43**data, executing actions on**

- commands

- description of, 174

- example of, 175-179

- implementing, 174

- reasons for using, 173

- usefulness of, 179

- iterators

- ATOM feeds, 182-185

- description of, 182

- example of, 183-185

- implementing, 182-183

- reasons for using, 181

- usefulness of, 185

- observers

- description of, 188

- example of, 189-191

- implementing, 188

- mailing lists, 189-191

- parts of, 187

- reasons for using, 188

- usefulness of, 191

- queues

- description of, 162

- example of, 163-170

- implementing, 163

- parts of, 161-162

- reasons for using, 162

- SQS, 165-166

- usefulness of, 170

database layers, n-tier web clusters, 198**databases (nonrelational), eventual consistency, 16****DB handlers, building blog application logic, 243-247****de-serialize method, User objects, 233-234****DELETE method, 61****deleting**

- buckets (S3), 77

- HTTP distributions in CloudFront, 80

- keys (S3), 77

- load balancers (ELB), 93

- messages from SQS queues, 82

posts (building blogs), 262-263

SQS queues, 83

volumes (EBS), 91

deploying applications, blog building example

base instances, starting, 272-273

images, bundling, 278

installing

 Apache, 276-277

 applications, 275-276

 software, 273-275

proxies, creating, 279

disposable instances, 83

Django, Jinja, 282

Domains, creating, 231

downloading

 boto python library, 35

 Marajo, 283

dumb terminals, 3

E

EBS (Elastic Block Storage), 88-91, 129

EC2 (Elastic Compute Cloud), 83

 connection objects, creating, 84

 images

 filtering, 86

 finding, 86

 running, 87

 searching for, 86

 instances, 87-92

 regions, viewing, 84

 reservation objects, 87

 security groups

 creating, 84

 opening ports to, 85

 removing rules from, 85

 SSH key-pairs, creating, 85

edit templates, creating, 292-296

editing posts (building blogs), 263-266

Elastic IP, 84

ELB (Elastic Load Balancing), 91-93, 155-158

 Auto-Scaling Groups, 47

 proxies, creating, 279

encoding video, 165-170

entry points (Setuptools), 177

environment (web), initializing, 282-283

ephemeral stores, 121

error messages

 409 Conflict, 61

 412 (Precondition Failed), 55

events

 observers

 description of, 188

 example of, 189-191

 implementing, 188

 mailing lists, 189-191

 parts of, 187

 reasons for using, 188

 usefulness of, 191

 SNS, 189

eventual consistency, 16, 31-32

external servers, querying, 138

F–G

Facades, 138

- description of, 148
- example of, 149–152
- implementing, 148–149
- mappers, 148
 - building, 149
 - example of, 149–152
 - request handling, 149–152
- requests, handling, 147
- usefulness of, 152
- using, reasons for, 148

failure, cloud providers, 27–28**filter layers, n-tier web clusters, 198****filtering images, EC2, 86****Flash browser-based clients, SaaS applications, 66****GET method, 58–59****Google**

- map/reduce clusters, 220
- PaaS, 23

Google App Engine, Jinja, 282**Google Cloud, 69**

- AppEngine, 108–110
- Google Storage, 111–112

Google Storage, 111–112**Graphics, clustering and, 5–6****Gui-Over-Database application framework, 141**

H

hadoop clusters, 174**ham/spam, marking comments as, 271–272****handlers, Marajo blog building example**

- creating, 286
- handlers directory, 283

hardware (commodity), 25**HEAD method, 61****Headers. See HTTP, headers****hiding messages in SQS queues, 82****horizontal scaling, 25****HTML browser-based clients, SaaS applications, 65–66****HTML5, 8–9****HTTP**

- CloudFront, 79–80
- headers
 - Accept headers, 56
 - Authorization headers, 56
 - If-Match headers, 55
 - If-Modified-Since headers, 55
 - If-Unmodified-Since headers, 56

REST

- collections, 58
- DELETE method, 61
- GET method, 58–59
- HEAD method, 61
- headers, 55–56
- OPTIONS method, 62
- POST method, 60
- properties, 58
- PUT method, 60
- resources, 58
- writing REST bodies, 57

HTTP Proxy

- blogs, building, 254-255
- Mod Proxy Balancer, 254
- Mod Proxy HTTP, 254

HTTPDate, 235**I****I/O throughput, performance and, 26****IaaS (Infrastructure as a Service), 23****If-Match headers, 55****If-Modified-Since headers, 55****If-Unmodified-Since headers, 56****images, 117-118**

- bundling, blog building example, 278

EC2

- filtering images, 86
- finding images, 86
- running images, 87
- searching for images, 86

prepackaged images

- boot process, 119
- description of, 120
- example of, 122-125
- implementing, 120-122
- reasons for using, 119
- usefulness of, 125

prototype images

- description of, 132
- example of, 133-135
- implementing, 132-133
- reasons for using, 131
- usefulness of, 135

pyami images, 133**singleton instances**

- description of, 128
- example of, 128-130
- implementing, 128
- reasons for using, 127
- usefulness of, 130

initializing web environments, Marajo**blog building example, 282-283****installing**

- boto python library, 35
- Marajo, 283
- Software, blog building example, 273-275

instances

- base instances, starting (application deployment strategies), 272-273

Elastic Block Storage backed instance, 129**prepackaged images**

- boot process, 119
- description of, 120
- example of, 122-125
- implementing, 120-122
- reasons for using, 119
- usefulness of, 125

prototype images, 131

- description of, 132
- example of, 133-135
- implementing, 132-133
- reasons for using, 131
- usefulness of, 135

pyami images, 133**singleton instances**

- description of, 128
- example of, 128-130

- implementing, 128
- reasons for using, 127
- usefulness of, 130

instances (EC2), 87-92

Internet speeds, cloud evolution, 5-6

IP (Elastic), 84

iterators, 181

- ATOM feeds, 182-185
- description of, 182
- example of, 183-185
- implementing, 182-183
- usefulness of, 185
- using, reasons for, 181

J-K-L

Java Applets, SaaS applications, 66

JavaScript browser-based clients, SaaS applications, 65-66

Jinja, 282

keys (S3)

- deleting, 77
- fetching contents of, 76
- listing, 75-76
- naming, 74

legacy patterns, 25-26

list templates, creating, 289-292

listeners. See observers

listing keys (S3), 75-76

listing posts, building blogs, 255-259

load balancers (ELB)

- creating, 92
- deleting, 93
- registering instances to, 92

local storage

- clouds, evolution of, 8-9
- HTML5, 8-9
- Mercurial, 8

locking clusters, semaphores and, 211

- parts of, 211
- using
 - description of, 212
 - example of, 213-218
 - implementing, 212
 - reasons for, 211
 - usefulness of, 218

locking processes, cloud-based application development, 15

logrotate, 121

loops (sqs read), 165

M

mailing lists, 189-191

mainframes, 3

map/reduce clusters

- description of, 220
- example of, 222-226
- implementing, 220
- parts of, 219
- usefulness of, 226
- using, reasons for, 220

mappers, 148

- building, 149
- example of, 149-152
- requests, handling, 149-152

Marajo

- blog building example
 - configuring applications, 287
 - creating handlers, 286

- creating resources, 284–286
- creating templates, 288–289
- custom templates, 289–296
- initializing web environment, 282–283
- running applications, 289
- development of, 282
- downloading, 283
- features of, 282
- installing, 283

Mercurial, 8, 142

messaging

- message queues, 15
- MHMessage Format, 165

MHMessage Format, 165

migrating to clouds, 26

mobile devices

- clouds, evolution of, 9–10
- Netflix and, 9

Mod Proxy Balancer, 254

Mod Proxy HTTP, 254

Multitasking, clouds and, 4

N

n-tier web clusters

- application layers, 198
- client layers, 197
- database layers, 198
- description of, 196
- example of, 198–210
- filter layers, 198
- implementing, 197–198
- parts of, 195–196
- representation layers, 198

- usefulness of, 210
- using, reasons for, 196

naming

- buckets (S3), 73
- keys (S3), 74

Netflix, mobile devices and, 9

next page links (ATOM feeds), 182

next tokens, 181, 184

nonrelational databases, eventual consistency, 16

O

oAuth authentication, Twitter, 141

objects

- commands
 - description of, 174
 - example of, 175–179
 - implementing, 174
 - reasons for using, 173
 - usefulness of, 179

REST, 58

observers

- description of, 188
- example of, 189–191
- implementing, 188
- mailing lists, 189–191
- parts of, 187
- usefulness of, 191
- using, reasons for, 188

Open Stack library, 17

OPTIONS method, 62

ORM (Object Relational Mapping) layers, 141

P

PaaS (Platform as a Service), 23

paging, 181

parallel computing, 11

parallel processing, cloud evolution, 10-11

partition tolerance, cloud services and, 30-31

PCs, cloud evolution, 3-4

performance

cloud providers, complaints against, 25

I/O throughput's effects on, 26

post handlers, building blog application logic, 248

POST method, 60

Post objects

data storage, building blogs, 230, 234-237

Reverse References, 235

posts (building blogs)

creating, 259-261

deleting, 262-263

editing, 263-266

listing, 255-259

prepackaged images

boot process, 119

description of, 120

implementing, 120-122

using

example of, 122-125

reasons for, 119

usefulness of, 125

presentation (blog building example), 253

comments, 266-269

adding, 270-271

marking as spam/ham, 271-272

viewing, 269-270

HTTP Proxy configuration, 254-255

posts

creating, 259-261

deleting, 262-263

editing, 263-266

listing, 255-259

processing

locking processes, cloud-based application development, 15

parallel processing, 10-11

speeds of, cloud evolution, 5-6

properties (objects), 58

prototype images

description of, 132

example of, 133-135

implementing, 132-133

usefulness of, 135

using, reasons for, 131

proxies

creating, 279

ELB, 155-158

proxy balancers, 137, 153

description of, 154

example of, 155-158

implementing, 154-155

usefulness of, 158

usefulness of, 158

using, reasons for, 153

PUT method, 60

pyami images, 133

Python

- boto python library
 - downloading, 35
 - installing, 35
 - requirements for, 34
 - software installations (application deployment strategies), 273-275
- python eggs, 199
- python paste, 241
- Reverend, Bayesian filtering systems, 250-253
- webob, 241
- WSGI, 241

Q

queries

- external servers, 138
- object queries, boto python library, 46
- SimpleDB, 43

queues

- description of, 162
- example of, 163-170
- implementing, 163
- message queues, 15
- parts of, 161-162
- SQS, 165-166
 - counting messages in queues, 82
 - creating messages in queues, 81-82
 - creating queues in, 80
 - default timeouts, 81

- deleting messages from queues, 82
- deleting queues, 83
- finding queues in, 81
- hiding messages in queues, 82
- reading messages from queues, 82
- usefulness of, 170
- using, reasons for, 162

R

Rackspace Cloud, 69

- CloudFiles, 112-113
- CloudServers, 113
- CloudSites, 113-114

rate limiting, Twitter, 141

RDS (Relational Database Service), 95-102

regions (Amazon Web Services), 70, 84

Relational Data Service, data storage in blogs, 231

representation layers, n-tier web clusters, 198

requests

- asynchronous requests, 137, 161
- commands
 - description of, 174
 - example of, 175-179
 - implementing, 174
 - reasons for using, 173
 - usefulness of, 179
- example of, 137
- Facades, 147
- handling requests. *See* architectures

- iterators
 - ATOM feeds, 182-185
 - description of, 182
 - example of, 183-185
 - implementing, 182-183
 - reasons for using, 181
 - usefulness of, 185
- proxy balancers, handling requests, 154
- queues
 - description of, 162
 - example of, 163-170
 - implementing, 163
 - parts of, 161-162
 - reasons for using, 162
 - SQS, 165-166
 - usefulness of, 170
- synchronous requests, 137
- reservation objects (EC2), 87**
- resources, 58**
 - deadlock, preventing, 212
 - Marajo blog building example
 - creating resources, 284-286
 - resources directory, 283
- REST (Representational State Transfer)**
 - bodies of, writing, 57
 - collections, 58
 - headers
 - Accept headers, 56
 - Authorization headers, 56
 - If-Match headers, 55
 - If-Modified-Since headers, 55
 - If-Unmodified-Since headers, 56

- methods
 - DELETE method, 61
 - GET method, 58-59
 - HEAD method, 61
 - OPTIONS method, 62
 - POST method, 60
 - PUT method, 60
- properties, 58
- resources, 58

Reverend, Bayesian filtering systems, 250-253

Reverse References, Post objects, 235

RRS (Reduced Redundancy Storage), 73

RTMP streams, CloudFront, 77

S

S3 (Simple Storage Service), 41

- billing for usage, 72

- buckets, 71

- accessing, 72

- deleting, 77

- keys, 74-77

- naming, 73

- sharing, 72

- uniqueness of, 73

- files

- accessing, 75

- getting contents as strings, 77

- making public, 76

- sending to S3, 75

- using objects directly, 77

- hierarchy of, 71

keys

- deleting, 77
- fetching contents of, 76
- listing, 75-76
- naming, 74

RSS, 73**signed URLs, creating, 76****SLA, 72****SaaS (Software as a Service). See also software****Amazon Web Services**

- account setup, 34
- logging into, 35
- viewing Access Credentials, 35

applications, developing

- accepting requests via REST, 53-62
- application layer, 47-52
- authorization layer, 62-64
- browser-based clients, 65-66
- .cfg files, 36-37
- client layer, 64-67
- data layer, 40-46
- determining application requirements, 39-40
- EC2 instances, 37
- file structures, 36-37
- HTTP headers, 53-56
- launching python instances, 38
- native applications, 66-67
- python logging modules, 38
- REST collections, 58
- REST methods, 58-62
- REST properties, 58
- REST resources, 58

setting up working environment, 36-38**SimpleDB, 38****writing REST bodies, 57****WSDL, 53****boto python library**

- creating objects, 45
- downloading, 35
- installing, 35
- querying objects, 46

business model, 33-34**subscriptions, 33-34****scalability**

- applications, 229
- cloud providers, 26
- eventual consistency, 31
- horizontal scaling, 25
- scaling out, 25
- scaling upward, 25

scaling out, 25**scaling upward, 25****scheduling systems (mainframe), 3****SDB, commands, 175****searches (image), EC2, 86****security groups, EC2**

- creating, 84
- ports, opening, 85
- rules, removing, 85

self-healing applications, 28**semaphores, locking clusters and**

- description of, 212
- example of, 213-218
- implementing, 212
- parts of, 211

- usefulness of, 218
- using, reasons for, 211
- serialize function, User objects, 232**
- servers**
 - application development,
 - cloud-based versus server-based, 11-13
 - external servers, querying, 138
 - sync servers, Mercurial and, 8
- Setuptools, entry points, 177**
- sharing buckets (S3), 72**
- signed URLs, 76**
- SimpleDB, 31, 43, 93-95**
- singleton instances**
 - description of, 128
 - example of, 128-130
 - implementing, 128
 - usefulness of, 130
 - using, reasons for, 127
- SLA (service level agreements), 72**
- SNS (Simple Notification Service), 102-105, 189**
- software. See also SaaS**
 - installing, blog building example, 273-275
 - paying for, 33
- spam filtering, building blogs**
 - Bayesian filtering systems, 250-253
 - comments, marking as spam/ham, 271-272
- SQS (Simple Queue Service), 165-166**
 - asynchronous procedure calls, 80
 - commands, 175
 - launching, 80

queues

- counting messages in, 82
- creating, 80
- creating messages in, 81-82
- default timeouts, 81
- deleting, 83
- deleting messages from, 82
- finding, 81
- hiding messages in, 82
- reading messages from, 82
- read loops, 165

SSH key-pairs, creating in EC2, 85

static directory, Marajo blog building example, 283

statistics (usage), obtaining via cloudFront, 78

storage

- blogs, building, 229
 - Comment objects, 237-240
 - database schema, 230
 - domains, creating, 231
 - Post objects, 230, 234-237
 - Relational Data Service, 231
 - User objects, 230-234
- cloud services, 23
- EBS (Elastic Block Storage), 88-91, 129
- Google Storage, 111-112
- local storage
 - clouds, evolution of, 8-9
 - Mercurial, 8
- shared storage systems, uploading data to, 15
- storage services, 24

strings, getting file objects as (S3), 77
subscriptions

Amazon Web Services, 34
 SaaS business model, 33–34

sync servers, Mercurial and, 8

synchronous requests, 137

syslog, 121

T

templates, Marajo blog building example

Creating, 288–289
 custom templates, 289–296
 edit templates, 292–296
 list templates, 289–292
 templates directory, 283

testing SimpleDB connectivity, SaaS applications, 38

thin clients, 3

threading clouds, evolution of, 10

transcoding video in multiple formats/definitions, 222–226

Twitter

application architectures, example of, 141–145
 OAuth authentication, 141
 rate limiting, 141

U–V

Ubuntu, application deployment strategies, 272–273

UNIX

logrotate, 121
 syslog, 121

Unsubscribe policy (CAN-SPAM), SNS, 189

updates

CloudFront, delivering updates via, 78
 instances (EC2), 90
 volumes (EBS), 89–90

URLs (signed), creating in S3, 76

usage statistics, obtaining via CloudFront, 78

User handlers, building application logic in blogs, 248

User objects

data storage, building blogs, 230–234
 de-serialize method, 233–234
 serialize function, 232

UUID

cloud-based application development, 14
 object relational mapping in SaaS applications, 45

video

encoding, 165–170
 transcoding in multiple formats/definitions, 222–226

viewing comments (building blogs), 269–270

volumes (EBS), 89–91

VPC (Virtual Private Cloud), 106–108

W–Z

web browsers

browser-based clients, 65–66
 thin clients as, 3

web environment, initializing, 282-283

weblogs. See blogs, building

webob, 241

**WSGI (Web Service Gateway Interface),
building blogs, 241-242**

Comment handlers, 249-250

DB handlers, 243-247

post handlers, 248

User handlers, 248

YAML applications, configuring, 287

zones (Amazon Web Services), 70