



Adriaan de Jonge

Essential App Engine

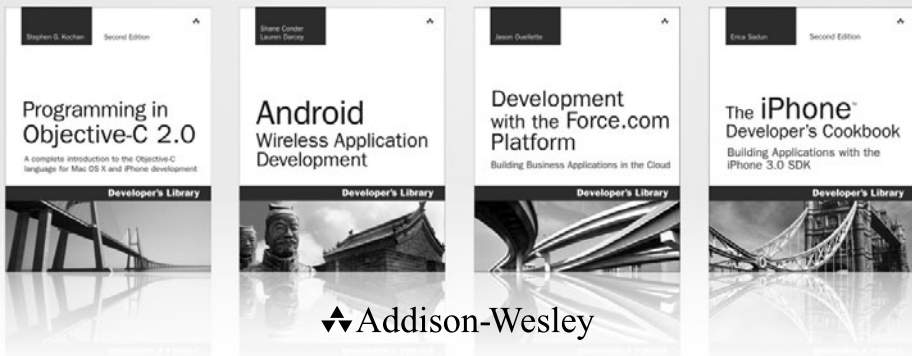
Building High-Performance Java Apps
with Google App Engine

Developer's Library



Essential App Engine

Developer's Library Series



Visit developers-library.com for a complete list of available products

The **Developer's Library Series** from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that's useful for other programmers.

Developer's Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

PEARSON

Addison-Wesley

Cisco Press

EXAM/CRAM

IBM Press

QUE

PRENTICE HALL

SAMS

Safari
Books Online

Essential App Engine

Building High-Performance Java Apps with Google App Engine

Adriaan de Jonge

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • 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

Jonge, Adriaan de, 1979-

Essential app engine : building high-performance Java apps with Google App engine /
Adriaan de Jonge.

p. cm.

Includes index.

ISBN 978-0-321-74263-6 (pbk. : alk. paper)

1. Computer software—Development. 2. Software architecture. 3. Java (Computer program language) 4. Google Apps. I. Title.

QA76.76.D47D425 2012

005.1—dc23

2011030789

Copyright © 2012 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-74263-6

ISBN-10: 0-321-74263-X

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, October 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Anna V. Popick

Copy Editor
Carol Lallier

Indexer
Jack Lewis

Proofreader
Kelli Brooks

Technical Reviewers
Joseph Annuzzi
Romin Irani
Alex Moffat

Editorial Assistant
Olivia Basegio

Cover Designer
Gary Adair

Compositor
LaurelTech



To everyone who is chasing their dreams...



This page intentionally left blank

Contents at a Glance

Introduction **xix**

Acknowledgments **xxvii**

About the Author **xxix**

I: An App Engine Overview 1

1 Setting Up a Development Environment **3**

2 Improving App Engine Performance **17**

II: Application Design Essentials 29

3 Understanding the Anatomy of a Google App Engine Application **31**

4 Data Modeling for the Google App Engine Datastore **45**

5 Designing Your Application **57**

III: User Interface Design Essentials 67

6 Presenting the User Interface with HTML5 **69**

7 Fine-Tuning the Layout Using CSS3 **85**

8 Adding Static Interactions Using JavaScript **99**

9 Adding Dynamic Interactions Using AJAX **113**

IV: Using Common App Engine APIs 127

10 Storing Data in the Datastore and Blobstore **129**

11 Sending and Receiving E-Mail **155**

12 Running Background Work with the Task Queue API and Cron **171**

13 Manipulating Images with the App Engine Image Service **187**

14 Optimizing Performance Using the Memory Cache **203**

- 15** Retrieving External Data Using URL Fetch **215**
- 16** Securing a Web Application Using Google Accounts, OpenID, and OAuth **229**
- 17** Sending and Receiving Messages Using XMPP **241**

V: Application Deployment 253

- 18** Improving the Development Process **255**
- 19** Assuring Quality Using Measuring Tools **263**
- 20** Selling Your Application **277**
- Index **289**

Contents

Introduction xix

Acknowledgments xxvii

About the Author xxix

I: An App Engine Overview 1

1 Setting Up a Development Environment 3

Working with Eclipse Tools 3

Installing Plugins in Eclipse 4

Starting a New App Engine Project 7

Starting the Development Server 9

Deploying to the Online App Engine 11

Deploying from the Command Line 14

Starting the Development Server Command Line 15

Deploying to the App Engine Command Line 15

Summary 16

2 Improving App Engine Performance 17

Performing in the Cloud 17

Comparing the App Engine to Traditional Web Applications 18

Optimizing Payments for Resources 18

Measuring the Cost of Class Loading 18

Timing a Servlet That Contains a Library 19

Timing a Servlet That Does Not Contain a Library 21

Reducing the Size of web.xml 22

Avoiding Cold Startups 24

Reserving Instances with Always On 24

Preloading Classes Using Warm-Up Requests 24

Handling Concurrent Requests with Thread-Safe Mode 25

Handling Memory Intensive Requests with Backends 25

| | |
|--|----|
| Improving Performance in General | 25 |
| Optimizing Your Data Model for Performance | 25 |
| Avoiding Redundant Processing Using Cache | 25 |
| Postponing Long-Running Tasks Using the Task Queue | 26 |
| Improving Page Load Performance in the Browser | 26 |
| Working with Asynchronous APIs | 26 |
| Optimizing Your Application before Deployment | 27 |
| Summary | 27 |

II: Application Design Essentials 29

3 Understanding the Anatomy of a Google App Engine Application 31

| | |
|--|----|
| Uploading Files for Dynamic Deployment | 31 |
| Setting Up the Directory Structure | 33 |
| Specifying Deployment Parameters | 34 |
| Specifying Repeating Tasks | 37 |
| Specifying Datastore Indexes | 38 |
| Blacklisting IP Ranges | 38 |
| Configuring Log Levels | 39 |
| Configuring Task Queues | 39 |
| Securing URLs | 40 |
| Configuring the Administration Panel | 41 |
| Setting Application Basics | 41 |
| Setting the Current Version | 42 |
| Adding Users | 42 |
| Enabling Billing | 43 |
| Summary | 44 |

4 Data Modeling for the Google App Engine Datastore 45

| | |
|-------------------------------------|----|
| Moving Away from Relational Storage | 45 |
| Denormalizing Data | 45 |
| Aggregating Data without Joins | 46 |
| Designing Schemaless Data | 47 |

| | |
|---|----|
| Modeling Data | 47 |
| Designing at a Micro Level | 47 |
| Choosing Properties | 48 |
| Separating Entities | 49 |
| Creating and Maintaining Relationships among Entities | 50 |
| Maintaining One-to-Many and Many-to-Many Relationships | 51 |
| Working with Data | 52 |
| Performing Transactions | 52 |
| Performing Queries | 53 |
| Creating Indexes | 53 |
| Upgrading When the Datastore Is Involved | 54 |
| Summary | 55 |

5 Designing Your Application 57

| | |
|----------------------------|----|
| Gathering Requirements | 57 |
| Choosing a Toolkit | 58 |
| Choosing a Framework | 58 |
| Choosing a Template Engine | 60 |
| Choosing Libraries | 60 |
| Making Design Choices | 61 |
| Modeling Data | 62 |
| Modeling URLs | 64 |
| Handling Page Flow | 65 |
| Summary | 65 |

III: User Interface Design Essentials 67

6 Presenting the User Interface with HTML5 69

| | |
|--|----|
| Introducing HTML5 | 69 |
| Using Basic HTML5 Elements | 70 |
| Drawing Images Using the Canvas | 72 |
| Dragging and Dropping Items into Pages | 74 |
| Improving Form Elements | 76 |

| | |
|--|------------|
| Detecting a User's Geolocation | 77 |
| Storing Data on the Client Side | 78 |
| Storing Data across Sessions | 78 |
| Storing Session Data | 80 |
| Querying Structured Data Using a Local SQL Database | 81 |
| Summary | 83 |
| 7 Fine-Tuning the Layout Using CSS3 | 85 |
| Selecting Elements Using CSS3 | 85 |
| Understanding Specificity Calculation | 85 |
| Using IDs | 86 |
| Selecting Classes | 88 |
| Selecting Pseudo-Classes | 88 |
| Selecting Attributes | 89 |
| Selecting Elements | 90 |
| Selecting Pseudo-Elements | 91 |
| Using New Graphical Effects in CSS3 | 92 |
| Rounding Edges | 94 |
| Using 2D Animations | 94 |
| Using 3D Animations | 96 |
| Summary | 98 |
| 8 Adding Static Interactions Using JavaScript | 99 |
| Setting Up a Simplistic Example | 99 |
| Cleaning Up HTML Using Unobtrusive JavaScript | 102 |
| Reducing JavaScript Dependence by Progressively Enhancing the HTML | 106 |
| Optimizing Performance Using Event Delegation | 109 |
| Avoiding Global Variables | 110 |
| Summary | 112 |
| 9 Adding Dynamic Interactions Using AJAX | 113 |
| Using Classic AJAX without Frameworks | 113 |
| Communicating with the Server Using XML | 114 |
| Communicating with the Server Using JSON | 116 |
| Communicating with the Server Using HTML | 118 |

| | |
|---------------------------------------|-----|
| Using Google App Engine's Channel API | 120 |
| Opening a Channel from the Server | 120 |
| Handling Messages on the Client | 122 |
| Summary | 125 |

IV: Using Common App Engine APIs 127

10 Storing Data in the Datastore and Blobstore 129

| | |
|--|-----|
| Processing Data Synchronously | 129 |
| Storing Data Synchronously | 130 |
| Querying Data Synchronously | 133 |
| Retrieving Data Synchronously | 135 |
| Processing Data Asynchronously | 136 |
| Storing Data Asynchronously | 137 |
| Querying Data Asynchronously | 139 |
| Retrieving Data Asynchronously | 140 |
| Setting Up Transactions | 141 |
| Using Multitenancy to Introduce Namespaces | 144 |
| Storing and Retrieving Large Files | 146 |
| Storing Large Files in the Blobstore | 146 |
| Querying for the Content of Blobstore | 149 |
| Retrieving Files from the Blobstore | 150 |
| Uploading Bulk Data Using the Remote API | 151 |
| Summary | 153 |

11 Sending and Receiving E-Mail 155

| | |
|--|-----|
| Sending Confirmation E-Mails with HTML and Attachments | 155 |
| Parameterizing the Mail Body | 158 |
| Securing the Servlet | 158 |
| Logging Sent Mails on the Development Server | 159 |
| Using the JavaMail API as an Alternative | 159 |
| Comparing the Low-Level API to JavaMail | 161 |
| Receiving E-Mail | 161 |
| Configuring the Servlet to Receive Mail | 161 |
| Implementing the Servlet to Store Received Mail | 162 |

| | |
|--|------------|
| Reading E-Mail without the JavaMail API | 165 |
| Failures | 167 |
| Considering Performance and Quota | 167 |
| How Long Does It Take to Send an E-Mail? | 167 |
| What Is the Overhead on a Cold Instance? | 168 |
| How Does the Mail Receiver Perform? | 168 |
| Summary | 169 |
| 12 Running Background Work with the Task Queue API and Cron | 171 |
| Task Queuing | 171 |
| Queuing Send Mails | 172 |
| Configuring Task Queues | 174 |
| Managing Quota | 174 |
| Specifying Additional Options | 175 |
| Taking Advantage of Task Queues | 179 |
| Scheduling Tasks Using Cron | 180 |
| Configuring Tasks Using cron.xml | 180 |
| Taking Advantage of Cron | 182 |
| Reading HTTP Headers | 182 |
| Summary | 185 |
| 13 Manipulating Images with the App Engine Image Service | 187 |
| Minimizing the Use of the Image API | 187 |
| Reading and Writing Images | 187 |
| Reading from User Input | 187 |
| Writing to the Datastore | 190 |
| Reading from the Datastore | 191 |
| Writing to User Output | 193 |
| Reading from a File | 193 |
| Performing Simple Manipulations | 195 |
| Creating Thumbnails of Large Images | 195 |
| Cropping Images | 197 |
| Rotating Images | 198 |
| Flipping Images | 198 |

Performing Advanced Manipulations 198
Summary 201

14 Optimizing Performance Using the Memory Cache 203

Using the Cache API for Basic Purposes 203
 Considering the Pitfalls of a Cache 203
 Caching String Values 204
Implementing a Caching Strategy 206
 Reducing App Engine Load Using ETag Headers 206
Working with Fine-Grained Cache 209
 Implementing Serializable 209
 Caching Query Results in Raw Entity Format 210
Maintaining a Cache 210
 Invalidating Cache Items 211
 Clearing the Cache 212
Using Other Cache Utility Methods 213
 Putting and Getting Multiple Values 213
 Registering Error Handlers 213
 Incrementing Values 213
Using JSR 107 as an Alternative API 214
Summary 214

15 Retrieving External Data Using URL Fetch 215

Reading URLs Using GET Requests 215
 Using the Standard URL Fetch API 215
 Using the Low-Level URL Fetch API 217
Reading Results 218
 Interpreting Results 218
 Writing to Memory Cache 219
 Writing to the Datastore 219
Adding Options to URL Fetch 219
 Controlling Timeouts 219
 Handling Exceptions Gracefully 221
Posting Form Data 223
Fetching URLs Asynchronously 224

| | |
|----------------------------|-----|
| Consuming Web Services | 226 |
| Accessing RESTful Services | 226 |
| Communicating with SOAP | 226 |
| Considering Security | 226 |
| Using HTTPS | 227 |
| Using Open Ports | 227 |
| Summary | 227 |

16 Securing a Web Application Using Google Accounts, OpenID, and OAuth 229

| | |
|---|-----|
| Authenticating Users with Google Accounts | 229 |
| Authenticating Users with OpenID | 232 |
| Providing Access to Third Parties Using OAuth | 235 |
| Securing URLs in web.xml | 237 |
| Enforcing Authentication | 238 |
| Enforcing Secure Protocols | 238 |
| Security Considerations | 239 |
| Validating Input | 239 |
| Configuring Multitenancy | 239 |
| Storing Personal Data | 240 |
| Summary | 240 |

17 Sending and Receiving Messages Using XMPP 241

| | |
|-------------------------------|-----|
| Sending Messages Using XMPP | 241 |
| Receiving Messages Using XMPP | 244 |
| Receiving Subscriptions | 246 |
| Receiving Presence | 249 |
| Summary | 251 |

V: Application Deployment 253

18 Improving the Development Process 255

| | |
|---|-----|
| Optimizing the Development Process for the Internet | 255 |
| Thinking Like a Project Manager | 256 |
| Reducing Overhead | 256 |
| Knowing Your End Goal | 256 |

| | |
|--|-----|
| Cutting Away Unnecessary Activities | 257 |
| Improving Functionality | 258 |
| Setting Priorities | 259 |
| Planning Iterations | 259 |
| Practicing Experiment-Driven Development | 260 |
| Making Changes Gradually | 260 |
| Measuring Quality | 260 |
| Optimizing Developer Productivity | 261 |
| Performing Rituals | 261 |
| Using New Programming Languages | 261 |
| Managing Time and Surroundings | 261 |
| Summary | 262 |

19 Assuring Quality Using Measuring Tools 263

| | |
|---|-----|
| Testing on a Production Environment | 263 |
| Putting the Added Value of Testing in Perspective | 263 |
| Performing a Sanity Check | 264 |
| Minimizing Damage from Failures | 264 |
| Thinking Differently about Usability | 265 |
| Choosing Functionality over Appearance | 265 |
| Optimizing Usability by Analyzing Analytics | 265 |
| Checking Availability with the Capabilities API | 265 |
| Logging Unexpected Behavior | 269 |
| Profiling Continuously on Production | 271 |
| Measuring User Response to Your Interface | 273 |
| Summary | 275 |

20 Selling Your Application 277

| | |
|-----------------------------------|-----|
| Determining How to Approach Sales | 277 |
| Knowing Your Audience | 277 |
| Reaching Your Audience | 278 |
| Making the News | 278 |
| Writing Articles | 278 |
| Blogging | 279 |
| Writing on Twitter | 280 |
| Publishing Facebook Pages | 281 |
| Connecting through Facebook Apps | 282 |

| | |
|--|------------|
| Advertizing on Google Apps Marketplace | 282 |
| Using AdWords | 284 |
| Optimizing Your App for Search Engines | 285 |
| Using Social Bookmarking Sites | 285 |
| Attracting Customers Using Mobile App Stores | 285 |
| Converting Prospects into Paying Customers | 286 |
| Handling the Payment Process | 286 |
| Summary | 287 |
| Index | 289 |

Introduction

A single hype is not enough to change the world. But multiple hypes together can change it as long as they are part of a bigger trend.

This book discusses more than one hyped technology: cloud computing, NoSQL, and HTML5. The technologies in this book combine well with other hyped technologies: functional languages (Scala) and connected devices (iPhone, iPad, Android).

The Internet is changing the world. That is old news, yes, but because it's old news, you may easily overlook the Internet's ongoing dynamics and influences. A good indicator that you are missing the cybership is if you are still stuck on Spring and Hibernate. Frameworks solving yesterday's problems are blocking the way to handle tomorrow's challenges.

The Google App Engine is a perfect fit with current Internet trends. Reading this book gives you a head start with upcoming technologies. This Introduction describes how both the App Engine and this book fit in the current trends.

Analyzing Internet Trends

To analyze the current Internet trends, you need to take a few steps back in time and see what has happened in the past two decades.

Starting in the Nineties

Let's start with the early 1990s. At first, the World Wide Web was used mostly to serve static HTML pages. The best way to serve a dynamic web application was to configure a `/cgi-bin` directory connecting to Perl scripts or binary programs that redirected the output to the web visitor. Web applications were nowhere near as mature as classic office applications. By the late nineties, though, developers were incorporating best practices from classic office automation into web applications, and the Internet soared with the dot-com bubble.

Switching to the New Millennium

In the early 2000s, web programmers realized that a Model-View-Controller pattern was not such a bad idea after all. And around 2005, Asynchronous JavaScript and XML (AJAX) helped make web applications more interactive. By 2008, web applications and office applications were on the same maturity level, sharing many of the same technologies, such as SQL databases and heavy application servers. Some UI libraries even tried to mimic classic Windows interfaces, with the ultimate goal of bringing a not-so-user-friendly interface concept into the browser.

Analyzing Current Developments

Right now, you can see the start of a trend in which Internet technology surpasses the maturity level of classic office automation. The frontrunners in Internet technology are critically investigating all parts of their systems and analyzing their designs for fit with the requirements of the current Internet environment. New technologies are being developed from scratch with the Internet's scalability requirements as a first priority.

It won't be long until office automation will have trouble keeping up with Internet technologies. That is the point where office automation will start adopting the best practices from the Internet instead of the other way around.

Replacing SQL Databases with NoSQL

Relational databases are one of the most widely used technologies in classic office automation. They are mature, well standardized, taught in most schools and universities, and available in all sizes. However, they were designed at a time when storage was still expensive, the number of users was limited to the number of employees in a single company, and the focus of their use was on transaction processing.

Relational databases do not scale well. They were designed as central storages operating efficiently enough to handle most of their work alone. In larger environments, their capabilities can be expanded using horizontally or vertically distributed databases or load-balanced setups that replicate data among multiple machines. Usually this functionality requires expensive software, machinery, and specialized knowledge, though, so at the end of the day, relational databases are still limited.

Switching to NoSQL with the Google App Engine Datastore

A common characteristic of NoSQL databases is high scalability. NoSQL databases are designed specifically with the requirements of the Internet in mind. To serve millions of visitors around the world in a few hundred milliseconds, you need functionality beyond that of relational databases. If you do not need to serve that many visitors, you may still consider relational databases because of their consistency and transactional integrity. You should choose NoSQL only if the advantages match your requirements.

Google App Engine offers the datastore as NoSQL storage. It allows you to store entities, each with a set of key-value pairs. A value can also consist of an array of values. Benefits of the App Engine offering are that you need not worry about system administration, and its APIs easily integrate with the rest of the platform.

When you start working with the App Engine datastore, you discover that NoSQL databases have additional advantages over the classic SQL offerings. The APIs are less awkward to use than JDBC APIs.

Moving Away from Object Relational Mapping

Object relational mapping has always been painful. Doing the mapping yourself is so cumbersome that it scares developers into using heavy and code-intensive frameworks

like Hibernate, Java Data Objects (JDO), and the Java Persistence API (JPA). Choosing not to map relational structures to objects is virtually impossible. It would imply keeping JDBC connections open longer than necessary.

NoSQL databases relieve you from the burden of object relational mapping. If you insist, you can still map your datastore's structures to Java objects. This does not always make sense though. This book shows many examples of datastore entities being directly passed to an HTML template. The result is clean, simple, and efficient code.

Considering Alternative NoSQL Solutions

Examples of other NoSQL databases are Amazon SimpleDB, Riak, Voldemort, Microsoft Trinity, Hadoop, Cassandra, CouchDB, MongoDB, Kyoto Cabinet, Hypertable, GraphDB, Redis, Google Pregel, and Google BigTable (the underlying platform of the App Engine's datastore). Each of these products has its own characteristics. Some are key-value storages, graph storages, document storages, or variants of these structures.

After reading this book and practicing with App Engine datastore, you should investigate the various NoSQL initiatives for your work on platforms other than the App Engine. Many of the advantages of the datastore on the App Engine platform can also be found outside the App Engine. This is all part of a larger trend, after all.

Computing in the Cloud

Cloud computing changes the way you write your applications. Classic enterprise applications usually optimize performance by taking a performance hit at startup time. If your application is restarted only once every few months, that can be an acceptable strategy. However, on some cloud platforms, including Google App Engine, your applications may be started and stopped multiple times an hour. This means that you should optimize your application to start up extremely fast, which may require throwing out all heavyweight frameworks, like Spring or Grails.

Maintaining Systems in the Cloud

Hosting applications in the cloud means hosting without worrying about the underlying infrastructure. In all cloud offerings, you pay only for what you use. This is especially interesting if your site experiences sudden high spikes in visitors. In classic setups, you require a machine park that is standing still most of the time, waiting for the exceptional spike when it really needs to work.

The advantage of the App Engine over other cloud initiatives is that it scales automatically. You need not give orders to start up additional instances of your application. If you are worried about controlling your budget, you can set a maximum on your every day expenses. This helps you prevent bankruptcy after a distributed denial-of-service (DDOS) attack.

The App Engine does not expose details of the underlying operating system to its users. Cloud services like Amazon Elastic Compute Cloud (EC2) and Microsoft Azure let

users maintain their own instance of an operating system. This is a trade-off between freedom and maintenance costs.

The App Engine could be characterized as a software developer's cloud platform, whereas EC2 could be characterized as a system administrator's cloud platform. Microsoft Azure is most interesting if your company is already running on a full Microsoft stack. It fits best with the .Net developer community, although it must be mentioned that Microsoft also targets Java developers with its Azure platform.

Make an informed decision about which platform you'll use before you start developing, because some lock-in is involved. Don't choose the App Engine just because it's Google or because you liked the cover of this book. Choose it because you want a well-integrated platform that relieves you from the burden of system administration and automatically scales to sudden changes in demand even while you sleep.

Connecting with Other Cloud Offerings

You can also consider cloud computing from a nontechnical perspective. When managers discuss cloud computing, they are usually talking about Google Apps rather than Google App Engine. Google Apps includes Google Docs, Gmail, Google Calendar, and Google Sites for Business. The App Engine is just a technical platform on which software vendors can host their applications. Managers may not be interested in such hosting. They are interested in the applications.

The Google Apps Marketplace helps software vendors sell applications that integrate well with Google Apps. The Google App Engine is the ideal platform for hosting applications that integrate with Google Apps. Hosting in Google's cloud may also help when selling your application to customers who already use Google Apps.

Adopting HTML5

HTML4 and XHTML1 have ruled the world for a long time. Now it is time to move on. The World Wide Web made a shift from serving documents to serving web applications. And even though documents will probably be served until the end of time, the real technical challenge is in serving user-friendly web applications.

Web interfaces are more easily understood by the average user than classic Windows interfaces. In operating systems, you can see a trend toward simplifying client-side interfaces to work similarly to web interfaces. Smart phones show similar advancements. Smart phone vendors are trying to keep up with the simplicity of Apple's iPhone.

HTML4 and XHTML1 have some limitations that quickly become awkward when using them to offer web applications. A lack of descriptive HTML element names is just a minor flaw that leads to overly complicated Cascading Style Sheet (CSS) files. HTML5 fixes this problem. More interesting are the additional JavaScript APIs offered with the HTML5 specification.

Using HTML5 offers many benefits. For example, consider the File Chooser dialog when uploading a file. HTML5 allows you to drag and drop files into your browser. You

can try this by adding an attachment in Gmail using drag and drop. Another example is the use of cookies or heavy server-side sessions. HTML5 offers session storage, local storage, and IndexedDB to store about 5MB of data on the client for later reuse. HTML5 allows you to make drawings on the client side using the Canvas.

Finally, the support for HTML5 on mobile and connected devices is better than you might expect. Some of the features of HTML5 are particularly useful on handheld devices. The lack of Flash support on iPhone and iPad is well compensated by HTML5. And possibly one of the most interesting features of HTML5 on a mobile device is the ability to ask for the user's location. If the user allows it, you can use it to customize search results to the things most relevant in that particular area.

Discussing Trends Out of Scope for This Book

Essential App Engine: Building High-Performance Java Apps with Google App Engine discusses some of the latest trends in cloud computing with NoSQL and HTML5. Some related trends are beyond the scope of this book, but with some additional reading, you can combine these trends with the technologies discussed here.

Serving Apps on Connected Devices

The examples in this book assume that the visitors are accessing the application using a web browser or a mobile browser. All examples target HTML, CSS, and JavaScript.

In addition to browsers, applications are increasingly served through platform-specific applications running on the iPhone, iPad, Android, Windows Phone, or BlackBerry. Numerous books on developing applications for these platforms are available.

From an App Engine perspective, requests from mobile applications are in many ways similar to AJAX requests made from browsers. You can serve JSON (JavaScript Object Notation) strings over a RESTful interface, providing the same data in a format that is easily read by the applications.

Moving to New JVM Languages

Java has been called the Cobol of the 21st century. Without arguing against that, the examples in this book are nevertheless in Java. A seeming trend away from the Java language does not necessarily imply moving away from the Java Virtual Machine (JVM). Most popular new languages like Scala and Clojure compile to JVM bytecode.

At this point, the Java language is still the largest language on the JVM platform. Despite its growing popularity, Scala has nowhere near the user base of Java yet. And even for those who are interested in other JVM languages, the Java language itself serves well as the lingua franca of JVM languages. This book demonstrates the Google App Engine API in a language-neutral way, independent of the heavy Java framework. Code examples in this book easily translate to Scala, Clojure, Groovy, JRuby, or Jython.

This Book's Target Audience

Essential App Engine is written for software developers and software architects.

For *software developers*, this book provides a hands-on approach to developing applications for the Google App Engine. It contains many simple, standalone code examples that demonstrate the concepts without distractions of unrelated code and frameworks. Software developers can modify the examples to use as working code, realizing their applications.

Software architects can read this book to get a general overview of the characteristics of the App Engine platform. In addition to the code examples, this book provides in-depth background knowledge of how the App Engine datastore differs from classic relational databases. It covers how you should change your design to get the best performance out of it. In addition, this book provides many pointers on how to change the way you design web applications to optimize their performance when hosted in the cloud.

Overview of This Book

This book contains twenty chapters divided into five parts. The order of the parts is consistent with a software development project that follows a design-first approach. You can read the chapters in a different order, though: Chapters are cross-referenced when more detailed background knowledge is desirable.

- **Part I, “An App Engine Overview,”** introduces you to the basics of the App Engine. It presents a discussion of performance characteristics and a practical guide to setting up your development environment so that you can continually address performance.
- **Part II, “Application Design Essentials,”** discusses all configuration options in the App Engine platform. It provides a design philosophy for modeling your data, targeting the Google App Engine datastore. And it discusses general technical design choices you should make before you start developing for the App Engine, such as whether or not to use Java Server Pages.
- **Part III, “User Interface Design Essentials,”** focuses on modern browser technology rather than on the App Engine itself. HTML5 and CSS3 are great companions when developing web applications in the cloud. The added possibilities in the browser help relieve the server from a lot of work and memory usage, ultimately lowering your usage costs while leveraging a responsive and user-friendly application to your client. In addition to discussion of HTML5 and CSS3, Part III provides an elaborate explanation of how to use JavaScript and AJAX to continue programming on the client side.
- **Part IV, “Using Common App Engine APIs,”** contains everything you need to know about the App Engine APIs. This includes the datastore, the Blobstore, the

Mail API, task scheduling, memory cache, URL retrieval, web application security, and XMPP messaging.

- **Part V, “Application Deployment,”** discusses how to improve your development process, optimize the quality of your web application, and sell it to potential customers.

The Essential App Engine Blog

Google provides frequent updates to the App Engine, adding new features and APIs, in response to popular demand. To keep you up to date, a companion website to this book, the Essential App Engine blog, is available at www.essentialappengine.com.

Check this website for the latest updates, the source code for this book, and additional code examples!

This page intentionally left blank

Acknowledgments

Writing a book is impossible without a strong and reliable support team.

First, I'd like to thank everybody at Addison-Wesley for all their help and support.

I owe special thanks to the following people:

- Trina MacDonald, for helping me through the process, providing practical tips on project planning, being my conscience for keeping the schedule, and knowing when to be patient and impatient at exactly the right times.
- Michael Thurston, for the thorough and detailed feedback on all texts and structure.
- Olivia Basegio, for all the work behind the scenes, keeping things running flawlessly at all times.
- The technical editors Joseph Annuzzi, Romin Irani, and Alex Moffat, each with their own specific area of interest and expertise. This book was greatly improved thanks to all their feedback, suggestions, and ideas, both high-level and in great detail.
- Carol Lallier, for making great improvements in the text while copy editing.

I'd like to thank everybody at ANWB who showed interest in the writing process of this book. You can hardly imagine the positive effect of your involvement.

Last but not least, great thanks go to all my family and friends. Thank you, first, simply for being my family and friends. And thank you for bearing with me throughout the process of writing this book—even if it sometimes meant I spent less time with you.

This page intentionally left blank

About the Author

Adriaan de Jonge is an online specialist in the Netherlands. He has worked in several roles: researcher, consultant, software architect, and author. He is not planning to settle down in a single role any time soon.

His areas of interest are Internet, gadgets, buzzwords, programming languages, and datastores—almost anything as long as it is new, lightweight, and challenging food for thought.

Adriaan works for ANWB, the Dutch association for tourism, traffic, and roadside assistance.

Chapter 2

Improving App Engine Performance

Throughout this book, a lot of attention is given to performance optimization. By improving performance, you get the added benefit of lowering the usage costs of your application when you surpass the App Engine's free quota. This chapter explains performance characteristics specific to the Google App Engine environment. It starts by discussing the process of starting and stopping instances in the cloud. The cost of starting an instance is demonstrated by showing the performance of a servlet using a third-party library compared to the performance of a plain vanilla servlet. This chapter also offers pointers for minimizing and, where possible, avoiding cold startups. Finally, it provides a high-level overview of performance-related topics you can find in other chapters in this book.

Performing in the Cloud

One of the unique selling points of cloud computing over traditional hosting is high scalability and flexibility when responding to changes in the demand of your application. The pricing model of cloud computing is especially convenient if you experience sudden high spikes in the number of visitors on a regular basis.

In the cloud, you pay for what you use. On the App Engine, this means that if your traffic is usually below Google's free daily quota and you have only incidental traffic spikes, you pay only for the computing power used during the days with high spikes. The advantage of cloud computing over having a physical machine park capable of handling high-traffic spikes is that you are not paying for machines that remain idle except during a traffic spike.

This flexibility also introduces a new challenge that might not be apparent at first sight. Responding to changes in demand means starting and stopping instances multiple times per hour. The time necessary to respond to a change in demand is directly related to the time necessary to start your web application. This means that your web application does not necessarily become flexible and scalable simply because it is deployed on the App Engine. You need to optimize your application to get the most out of the specific circumstances of running on the Google App Engine.

Comparing the App Engine to Traditional Web Applications

Whereas the lifetime of a typical App Engine instance is measured in minutes and hours, the lifetime of a traditional web application instance is measured in weeks or months.

Traditional web application here means a web application running on a physical machine that you maintain yourself rather than an application running in the cloud.

One of the most common approaches to optimizing the performance of a traditional web application is to take a performance hit on startup of the instance. For example, if you load a lot of classes and data into memory during startup, you can save loading time while processing the actual user requests because starting and stopping an application instance is unrelated to handling a request.

Taking a performance hit during the startup of a new instance is not such a good idea, though, if a website visitor is waiting while your application is starting. You may lose a visitor every time a new instance is started.

In addition, the scalability requirements of the App Engine ask for different storage strategies. Most traditional web applications are based on relational databases. Strategies for optimal usage of a relational database can sometimes be catastrophic when applied to NoSQL storages like the Google App Engine datastore.

As a result, web application frameworks originally designed for use with software stacks can lead to bad results when used on the App Engine without consideration.

Optimizing Payments for Resources

On the App Engine, you pay for the resources you use. This means that optimizing your application to use fewer resources also leads to cost reductions.

On the App Engine, some resources are more expensive than others. The optimal usage versus cost ratio depends on the characteristics of your application. How much data do you store? How much traffic is generated by your visitors? How is the traffic distributed over the total data set? How much data processing is involved? How is the number of visitors distributed over time?

When you consider these questions and look at the current pricing tables on Google's site, you quickly find that you may have an optimization challenge. Take a look on <http://code.google.com/appengine/docs/billing.html> for more information.

Although there is no silver bullet for an optimal cost reduction, this book aims to give you the most control over the performance and costs of your web application.

Measuring the Cost of Class Loading

Every library or framework you introduce brings lots of additional classes to load at startup. For this reason, this book introduces only three third-party JARs to help with the code examples: Commons FileUpload, StringTemplate, and ANTLR. Commons FileUpload is used to process form submits with files as content. StringTemplate is used as a template language to generate output for the visitors, and it can also be used to generate text for an e-mail. ANTLR stands for Another Tool for Language Recognition and is a dependency of StringTemplate.

To show you the cost of class loading, this chapter investigates the startup time of the App Engine instance with `StringTemplate` and without `StringTemplate`. In addition, there is a startup time comparison between a `web.xml` file of roughly 400 lines and a `web.xml` of 21 lines.

Timing a Servlet That Contains a Library

Listing 2.1 shows a very simple servlet that processes a template using the `StringTemplate` framework and shows “Hello, World” in the browser window.

Listing 2.1 Writing Hello World with `StringTemplate`

```
01 package com.appspot.template;
02
03 import java.io.IOException;
04
05 import javax.servlet.ServletException;
06 import javax.servlet.http.HttpServlet;
07 import javax.servlet.http.HttpServletRequest;
08 import javax.servlet.http.HttpServletResponse;
09
10 import org.antlr.stringtemplate.StringTemplate;
11 import org.antlr.stringtemplate.StringTemplateGroup;
12
13 public class StringTemplateServlet extends HttpServlet {
14
15     protected void doGet(HttpServletRequest request,
16                          HttpServletResponse response)
17         throws ServletException, IOException {
18         long startTime = System.currentTimeMillis();
19
20         StringTemplateGroup group = new StringTemplateGroup("xhtml",
21                 "WEB-INF/templates/xhtml");
22         StringTemplate hello = group.getInstanceOf("hello-world");
23         hello.setAttribute("name", "World");
24         response.getWriter().write(hello.toString());
25
26         long diff = System.currentTimeMillis() - startTime;
27         response.getWriter().write("time: " + diff);
28
29     }
30 }
```

Lines 18 and 26 process the timer, while the code loading the `StringTemplate` and ANTLR JARs are on lines 20 through 24.

Writing the resulting time at the bottom of the HTML (line 27) is not really elegant, but it works sufficiently for the simple timer required in this example.

Line 22 refers to an external file with an HTML template. This template is shown in Listing 2.2.

Listing 2.2 Setting Up the HTML Template for StringTemplate

```
01 <html>
02   <head>
03     <title>Test</title>
04   </head>
05   <body>
06     Hello, $name$ from a file!
07   </body>
08 </html>
```

Line 6 processes the attribute provided in line 23 of Listing 2.1. The rest of the HTML template should not require any explanation. The resulting screen just after a new instance is launched is displayed in Figure 2.1.

Reloading the same servlet when the instance is already started is a lot faster. Processing the StringTemplate takes 10 to 15 milliseconds on subsequent requests.

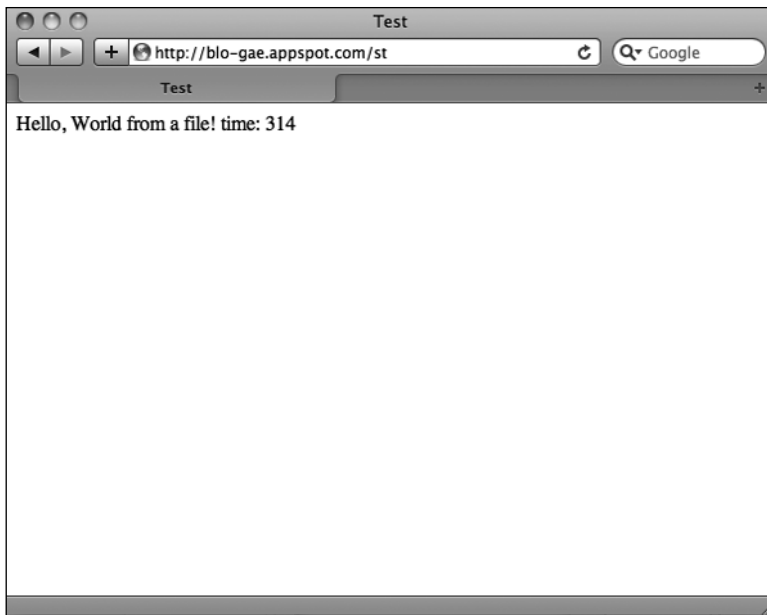


Figure 2.1 Displaying the resulting time in the browser screen with StringTemplate.

Timing a Servlet That Does Not Contain a Library

Writing Hello World to a browser screen is simple enough to do without a library like `StringTemplate`. If you modify the code to write Hello World directly to the browser, you get a servlet as shown in Listing 2.3.

Listing 2.3 Writing Hello World without `StringTemplate`

```
01 package com.appspot.template;
02
03 import java.io.IOException;
04
05 import javax.servlet.ServletException;
06 import javax.servlet.http.HttpServlet;
07 import javax.servlet.http.HttpServletRequest;
08 import javax.servlet.http.HttpServletResponse;
09
10 public class StringTemplateServlet extends HttpServlet {
11
12     protected void doGet(HttpServletRequest request,
13                          HttpServletResponse response)
14         throws ServletException, IOException {
15
16         long startTime = System.currentTimeMillis();
17
18         response.getWriter().write("Hello World without ST! ");
19
20         long diff = System.currentTimeMillis() - startTime;
21         response.getWriter().write("time: " + diff);
22
23     }
24 }
```

The only difference is in line 18. To avoid wasting too much code, the HTML is left out. Seven short lines of HTML do not have a significant influence on the loading time: they account for less than a millisecond.

Figure 2.2 shows the browser window loading the servlet from Listing 2.3 while starting a new instance. The decrease in loading time is substantial!

If loading the `StringTemplate` library increases the loading time of a new App Engine instance by 300 milliseconds, then why not switch to `FreeMarker`, `Velocity`, or `Java Server Pages (JSP)`, you might ask. Or perhaps you know another template engine not mentioned here. You are encouraged to investigate and find out for yourself which library has the most efficient loading times on cold startup.

For any other library or framework you'd like to introduce, you should first investigate what the effect is on the total load time. Adding an additional JAR is always a big step.



Figure 2.2 Displaying the resulting time in the browser screen without StringTemplate.

Reducing the Size of web.xml

Explicit changes like adding JARs are relatively simple to manage. More tricky is making changes more gradually over time. For example, this book is full of servlets. As servlets were added, the web.xml file grew. At the end of the writing, the web.xml file contained more than 400 lines of configuration setting up all the examples demonstrated in the book.

The number of servlets declared in web.xml has a significant influence on the class loading time. To test the difference, the web.xml was reduced to minimal size, as shown in Listing 2.4. Just a single servlet is included—the servlet from Listings 2.1 and 2.3.

Listing 2.4 Reducing web.xml to an Absolute Minimum

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03     xmlns="http://java.sun.com/xml/ns/javaee"
04     xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
05     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06         http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
07     version="2.5">
08
```

Listing 2.4 Reducing web.xml to an Absolute Minimum (Continued)

```

09  <!-- Template -->
10  <servlet>
11    <servlet-name>StringTemplateServlet</servlet-name>
12    <servlet-class>
13      com.appspot.template.StringTemplateServlet
14    </servlet-class>
15  </servlet>
16  <servlet-mapping>
17    <servlet-name>StringTemplateServlet</servlet-name>
18    <url-pattern>/st</url-pattern>
19  </servlet-mapping>
20
21 </web-app>

```

Take a look at the log files before and after the web.xml size reduction. Figure 2.3 shows the difference in CPU usage for both scenarios.

As you can see, the difference in load time on cold startup is significant. This is an indication that you should be careful with the number of servlets you declare in a web application. On the other hand, one very large servlet is unlikely to perform much better

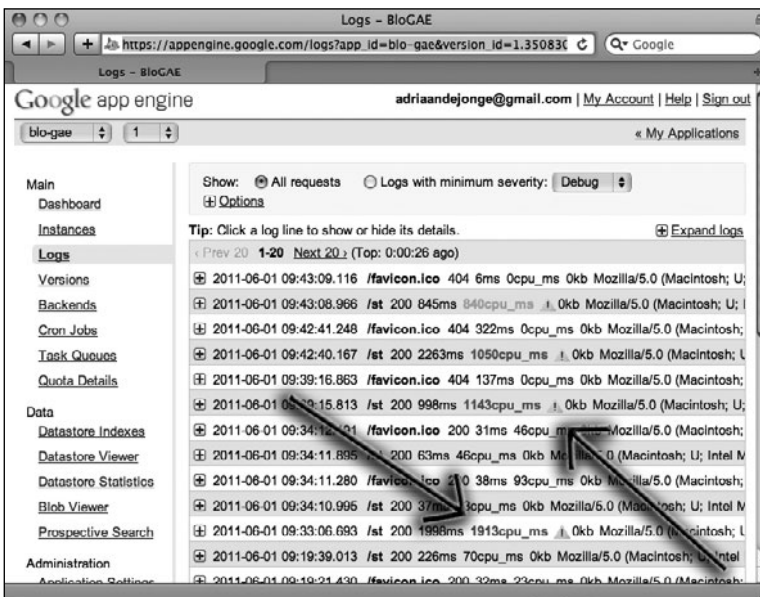


Figure 2.3 Displaying the logged CPU times before and after a web.xml reduction.

than several smaller ones, so you must consider the trade-off. How do you divide your code over a number of servlets with the least class loading overhead? Again, there is no silver bullet for doing so. The important thing is that you think about this trade-off in your specific situation.

Avoiding Cold Startups

In the early days of the Google App Engine, any request could lead to a new instance being launched. For applications with low traffic, there was a high risk of long response times on the first request by a visitor, especially if the application was not optimized for fast cold startups.

Only high-traffic applications with a relative constant load could serve a large percentage of users without confronting them with longer response times. But even those would lose a few visitors with instance starts and stops.

Later, Google added new features for paying customers that help avoid longer response times. It should be noted that these strategies may fail when the application experiences very sudden spikes in traffic.

Reserving Instances with Always On

Paying customers can hire instances that are never turned off. This solves the problem of low-traffic applications, where almost every visit leads to an instance being launched.

The Always On instances are supplemented with dynamic instances when the demand exceeds the capabilities of the available Always On instances. This means that just switching to Always On does not completely fix the problem with long responses on cold startups.

Always On can be configured in the admin console, as described in Google's documentation on <http://code.google.com/appengine/docs/adminconsole/instances.html>.

Preloading Classes Using Warm-Up Requests

When at least one instance is running, either Always On or dynamic instances, the App Engine can sometimes predict when a new instance will be required.

As long as you haven't explicitly turned off warm-up requests in the `appengine-web.xml` configuration file, the App Engine can send a request to `/_ah/warmup` sometime before a new instance is required. You can configure your own servlet to listen on that address and make sure that classes and other data are preloaded before a visitor starts accessing that instance.

Warm-up requests do not work when no instances are running. They do not add much value for low-traffic applications unless Always On is used.

Even with instances running, warm-up requests do not always work. The App Engine is not always capable of predicting traffic in advance.

More information on warm-up requests is found on <http://code.google.com/appengine/docs/adminconsole/instances.html>.

Handling Concurrent Requests with Thread-Safe Mode

By default, an instance handles only a single request at a time. If an instance takes long to respond and there are other requests at the same time, the App Engine launches additional instances to handle the rest of the traffic.

In some cases, loading new instances can be avoided by allowing concurrent requests. This requires you to develop thread-safe servlets. More information on thread-safe mode is found on <http://code.google.com/appengine/docs/java/config/appconfig.html>.

Handling Memory Intensive Requests with Backends

In addition to Always On instances, you can purchase, for a higher fee, specialized instances that are optimized for handling requests of a backend nature—that is, requests that require longer than 30 seconds to finish. Another characteristic of backend applications is higher memory consumption.

More information on backend instances can be found on Google’s website at <http://code.google.com/appengine/docs/java/backends/>.

Improving Performance in General

The subtitle of this book is *Building High-Performance Java Apps with Google App Engine* because this book focuses on performance optimization more than do other books. This section provides a general overview of possibilities for performance optimization.

Optimizing Your Data Model for Performance

If you model your data for the App Engine datastore the same way you model your data for a relational database, you can be certain that you will run into performance problems at some point. The way the App Engine datastore divides data over multiple machines in the cloud is fundamentally different from the way a relational database stores data on disk. In many cases, you need to do the exact opposite of what you are used to doing. For example, you need to denormalize your data instead of normalizing it.

Because you can store arrays of data, there is less need of relationships between tables, although you should be cautious if you feel the need to index the array, because the size of your total index may explode.

You should consider the need for transactions before you set up your data model. Transactions require entity groups, and larger entity groups may harm scalability.

Chapter 4, “Data Modeling for the Google App Engine Datastore,” presents a detailed discussion of datastore characteristics. Using the APIs is demonstrated in Chapter 10, “Storing Data in the Datastore and Blobstore.”

Avoiding Redundant Processing Using Cache

Many time-consuming tasks are done repeatedly for subsequent requests—think of tasks that require gathering data or processing intensive calculations. The same processing might be repeated for a single visitor or for multiple visitors.

Proper caching can help avoid repetitive processes. This book explains both fine-grained caching using memcache and page-level caching on the Internet. See Chapter 14, “Optimizing Performance Using the Memory Cache,” for in-depth information.

Postponing Long-Running Tasks Using the Task Queue

In many cases, high responsiveness is more important than high performance. Responding quickly to a visitor’s request can sometimes be done by postponing the actual work. As long as the visitor can trust that the work will be done eventually, he or she will be pleased with the quick response.

The Task Queue API can be used in multiple ways. You can preschedule tasks at regular intervals, or you can post tasks to the queue on demand. Both methods can help improve performance and responsiveness.

Details on Task Queue are discussed in Chapter 12, “Running Background Work with the Task Queue API and Cron.”

Improving Page Load Performance in the Browser

A high-performing server is practically useless if the page loading in the browser ruins the total response time. For example, if your HTML is full of useless elements, classes, and IDs, your Cascading Style Sheet (CSS) file beats the size of an average phone book, and you reach a megabyte of JavaScript files, all server-side efforts are lost. You could make it even worse by adding one or more Flash files in your page. But then you are clearly working in the wrong direction.

With HTML5 and CSS3, you hardly need Flash anymore except, perhaps, for an incidental video player being used until HTML5 videos are sufficiently mature. The newly added elements in HTML5 may help you downsize your CSS files. The less specific your CSS file, the easier it is to maintain.

The way you load your JavaScript has a large impact on the page load time. Loading JavaScript unobtrusively at the bottom of the page allows the rest of the page to render before the JavaScript is interpreted. This improves the responsiveness to the visitor.

Part III, “User Interface Design Essentials,” covers HTML5, CSS3, JavaScript, and AJAX, providing details on browser optimization from a technical perspective.

Working with Asynchronous APIs

Page loading generally does not entail heavy data processing. Mostly it consists of waiting for services such as the datastore to respond. If you know in advance that you need to make multiple backend requests and the backend requests are independent of each other, you can work with asynchronous APIs.

One of the most important asynchronous APIs is described in Chapter 10, “Storing Data in the Datastore and Blobstore.”

Optimizing Your Application before Deployment

Some performance optimizations are a result of planning and designing. The more effective performance improvements usually result from careful experimentation and measurements.

You can profile calls to Google's backend services using AppStats. Most of the overhead in an average App Engine application is in the backend calls. If you do a lot of heavy lifting in your own code, you are encouraged to profile this code and optimize where possible.

AppStats is explained in Chapter 19, "Assuring Quality Using Measuring Tools."

Summary

Cloud solutions, and specifically the Google App Engine, are designed for scalability and flexible usage from scratch. However, in the case of the Google App Engine, this design may mean that some classic performance optimization strategies are counterproductive. This chapter focused on cold startup time and why you should avoid cold startups when possible. It also discussed the overhead of frameworks and libraries—also to be avoided when possible. The end of this chapter presented a few performance questions with cross references to the chapters where you can find the answers.

Index

Numbers

2D animations, in CSS3, 94–96

3D animations, in CSS3, 96–98

A

A/B testing, 260

Abstraction, frameworks adding level of, 58–59

Access control. *See also* Authentication

granting user access to applications,
42–43

providing third party access, 235–237

restricting access at URL level, 238

securing server access, 227

AddThis, 285

Admin interface, logging messages to,
269–270

Administration panel, configuration options
in, 41–43

AdWords, 284

Aggregating data, without joins, 46

AJAX (Asynchronous JavaScript)

Channel API and, 120, 124–125

communicating with server using
HTML, 118–119

communicating with server using
JSON, 116–118

communicating with server using
XML, 115–116

handling messages on client, 122

handling page flow, 65

- overview of, 113
- returning tokens to visitors, 123–124
- sending messages to channels, 121–122
- setting up HTML for AJAX examples, 114
- setting up new channel from server, 120–121
- summary, 125
- without frameworks, 113

Always On instances, for avoiding cold startups, 24

Animation

- 2D animations in CSS3, 94–96
- 3D animations in CSS3, 96–98

Ant build tool, setting up directory structure with, 34

ANTLR (Another Tool for Language Recognition), 18, 61

Apache Commons File Upload. See Commons File Upload

Apache mod_cache, 206

APIs

- asynchronous. *See* Asynchronous APIs
- for background work. *See* Task Queue API
- for blobstores. *See* Blobstores
- for capabilities. *See* Capabilities API
- for channels. *See* Channel API
- for datastores. *See* Datastores
- for e-mail. *See* E-mail
- Fluent API, 175
- for geolocation, 77–78
- for Google Accounts. *See* Google Accounts API
- for image manipulating. *See* Image API
- for JavaMail. *See* JavaMail API
- for JavaScript, 70
- for local storage, 78–80

- low-level. *See* Low-level API
- for memory cache. *See* JSR 107 JCache API; low-level memory cache (memcache) API
- Objectify API, 129
- Pull API, 184–185
- Remote API, 151–153
- REST API, 184–185
- for retrieving external data. *See* URL Fetch API
- for sending messages. *See* XMPP (Extensible Messaging and Presence Protocol)
- for session storage, 80–81
- SimpleDS API, 129
- synchronous. *See* Synchronous APIs
- for task scheduling. *See* Cron
- Twig Persist API, 129
- for web application security. *See* Security

App stores, 285–286

appengine-web.xml

- configuring inbound e-mail, 162
- configuring XMPP services, 245
- specifying configuration parameters, 34–35
- specifying system properties, 36

Application anatomy

- overview of, 31
- summary, 43–44

application element, specifying deployment parameters, 34

Application Settings screen, Administration panel, 41

Applications

- deploying. *See* Deployment
- designing. *See* Designing applications
- registering, 3
- selling. *See* Selling applications

state and stateless, 78

AppStats, 263, 271–272

Architecture, cutting away unnecessary activities, 257

Articles, minimizing on page load, 108

Articles, writing for reaching audience, 278–279

Asynchronous APIs

processing, 136

querying, 139–140

retrieving, 140–141

retrieving URL Fetch responses, 224–226

storing, 137–139

working with, 26

Asynchronous JavaScript. See AJAX (Asynchronous JavaScript)

Attachments

security constraints and, 159

sending confirmation emails with HTML and attachments, 155–158

Attributes, selecting in CSS3, 89–90

Audience, knowing, 277–278

Audience, reaching

blogging, 279–280

Facebook, 281–282

Google AdWords, 284

Google Apps marketplace, 282–283

making news and writing articles, 278–279

mobile app stores, 285–286

overview of, 278

search engine optimization, 285

social bookmarking, 285

Twitter, 280–281

Authentication

enforcing, 237–238

with Google Accounts, 229–232

with OAuth, 226

with OpenID, 232–235

of owner of App Engine instances, 13

of third parties, 235–237

Authorization

with Google Accounts, 231

with OAuth, 235

Availability, checking, 265–269

B

Backends, for handling memory intensive requests, 25

Background operations. See Cron; Task Queue API

BigTable, Google, 45

Billing

for blobstores, 147

enabling, 43

Blacklisting IP ranges, 38–39

Blobs

querying metadata for available, 149–150

storing, 147–148

Blobstores

compared with datastore, 191

enabling billing for, 147

overview of, 146

querying content of, 149–150

retrieving files from, 150–151

storing large files in, 146–148

summary, 153

uploading file to, 148–149

Blogger, 279

Blogs/blogging

Essential App Engine blog, xxv

posts. *See* Posts

reaching your audience, 279–280

border-radius, rounding edges in CSS3, 94

Browser Cache, with ETag headers, 206–208

Byte array payload

- reading in task servlet, 178–179
- specifying task payloads, 178

C

Cache

- avoiding redundant processing, 25–26
- JSR 107 JCache API, 214
- memcache API. *See* Low-level memory cache (memcache) API
- writing results of URL Fetch to, 219

Cache keys, 205–206

Canvas element

- compared with Image API, 187
- drawing images with, 72–74

Capabilities API

- checking availability of App Engine services, 265–267
- displaying capabilities in HTML, 268–269
- preventing server errors, 263

Cascading Style Sheets. *See* CSS3 (Cascading Style Sheets)

Change management, in development process, 260

Channel API

- example, 124–125
- handling messages on client, 122
- overview of, 120
- returning tokens to visitors, 123–124
- sending messages to channels, 121–122
- setting up new channel from server, 120–121

Chat

- message types in XMPP, 244
- XMPP and, 241

Child elements, storing in transactions, 142

CIDR (Classless Interdomain Routing), 38

Class white list, JRE (Java Runtime Environment), 215

Classes

- measuring cost of class loading, 18–19
- preloading using warm-up requests, 24
- removing undesirable, 107
- selecting in CSS3, 88

Classless Interdomain Routing (CIDR), 38

click event listeners, 109–110

Clients/client side

- communicating with Remote API from, 152–153
- data storage on, 78–81
- message handling on, 122
- sending XMPP messages to Google Talk client, 242–243

Cloud computing

- application development and, 255
- maintaining system in the cloud, xxi–xxii
- overview of, xxi
- performance optimization and, 17–18

Code, JavaScript. *See* JavaScript

Cold startups

- avoiding, 24–25
- e-mail and, 168

Command-line tools

- for deploying directory structure, 14–15
- for starting development server, 15

Comments, in data modeling, 62–63

Commons File Upload

- libraries required in application design, 61
- reading images from user input, 190
- third-party JARs, 18

Composite images, 198–201

Configuration

- of Administration panel, 41–43
- blacklisting IP ranges, 38–39
- crontab configuration file, 180
- of log levels, 39
- of multitenancy, 239
- parameters in `appengine-web.xml`, 34–35
- of Remote API on server, 151–152
- securing URLs, 40–41
- of services, 37
- servlet to receive mail, 161–162
- setting up directory structure, 33–34
- specifying datastore indexes, 38
- specifying deployment parameters, 34–37
- specifying repeating tasks, 37
- of task queues, 39, 174
- of tasks using `cron.xml`, 180–182
- uploading files for dynamic deployment, 31–32
- of URLs with wild card in `web.xml`, 248–249
- of XMPP services, 245

Confirmation e-mails

- sending generally, 155–158
- sending with JavaMail API, 159–161

Console view, 10**Container objects, reducing number of global variables and, 111–112****Cookies, 78****Cron**

- benefits of, 182
- configuring, 37
- jobs, 32
- overview of, 180
- reading HTTP headers, 182–185
- setting up multiple tasks at different intervals, 180–181

- specifying repeating tasks, 37
- specifying start times, 181–182
- summary, 182–185

Crontab configuration file, in UNIX, 180**`cron.xml`**

- configuring tasks, 180–182
- function of, 180
- specifying repeating tasks, 37

Cropping images, 197**CSS3 (Cascading Style Sheets)**

- 2D animations in, 94–96
- 3D animations in, 96–98
- attribute selection, 89–90
- class selection, 88
- element selection, 90–91
- extending to allow minimization of blog posts, 101–102
- ID selectors, 86–88
- improving page load performance, 26
- new graphical effects in, 92–94
- overview of, 85
- pseudo-class selection, 88–89
- pseudo-element selection, 91
- rounding edges, 94
- specificity calculation, 85–86
- summary, 96–98

Customers

- converting prospects into, 286
- knowing your audience, 277–278
- reaching your audience. *See* Audience, reaching

D

Data models

- aggregating data without joins, 46
- choosing properties, 48–49

- creating indexes, 53–54
- creating/maintaining entity relationships, 50–52
- denormalizing data, 45–46
- designing applications, 62–63
- designing data at micro level, 47–48
- designing schemaless data, 47
- moving away from relational storage, 45
- optimizing performance with, 25
- overview of, 45
- performing queries, 53
- performing transactions, 52–53
- separating entities, 49–50
- summary, 54
- upgrades involving datastores, 54

Data replication, 46**Data storage**

- across sessions, 78–80
- securing personal data, 239
- of session data, 80–81

Data transfer objects, 257**DataNucleus, 34****Datastores**

- compared with blobstores, 191
- moving away from relational storage, 45
- multitenancy, 144–146
- NoSQL databases and, xx
- presenting HTML links to posts, 134–135
- processing data asynchronously, 136
- processing data synchronously, 129–130
- querying data asynchronously, 139–140
- querying data synchronously, 133–134
- reading images from, 191–193
- retrieving data asynchronously, 140–141
- retrieving data synchronously, 135–136
- setting up transactions, 141–144
- specifying indexes, 38

- storing data asynchronously, 137–139
- storing data synchronously, 130–133
- storing received XMPP messages in, 244–246
- storing XMPP presence notifications in, 249–251
- summary, 153
- upgrades involving, 54
- uploading bulk data, 151–153
- writing images to, 190–191
- writing results of URL Fetch to, 219

Date

- date input in BlackBerries and Safari browsers, 76
- scheduling tasks, 176

Defensive programming, 204**Delaying tasks, 176****Del.icio.us, 285****Denormalizing data, 45–46****Dependency injection, 257****Deployment**

- of applications from development server to App Engine, 15
- of applications to online App Engine, 11–14
- development process and. *See* Development process
- of directory structure, 14–15
- optimizing applications prior to, 27
- quality assurance and. *See* Quality assurance
- sales and. *See* Selling applications
- specifying application deployment parameters, 34–37
- uploading files for dynamic deployment, 31–32

Design details, cutting away unnecessary activities, 257**Designing applications**

- choosing framework, 58–59
- choosing libraries, 60–61
- choosing template engine, 60
- data model for, 62–63
- handling page flow, 65
- overview of, 57
- requirements gathering, 57–58
- summary, 65
- URL model for, 64–65
- working without framework, 59–60

Designing data

- at micro level, 47–48
- schemaless data, 47

Developers

- commercial orientation often lacking in, 256
- optimizing productivity of, 261–262

Development environment, setting up

- deploying application from development server to App Engine, 15
- deploying application to online App Engine, 11–14
- deploying directory structure, 14–15
- overview of, 3
- starting development server, 9–11, 15
- starting new App Engine project, 7–9
- summary, 15
- working with Eclipse tools, 3–7

Development process

- eliminating unnecessary activities, 257–258
- end goal in, 256–257
- experiment-driven development, 260
- frameworks in, 58
- gradual change in, 260
- improving functionality, 258
- optimizing for Internet, 255–256
- optimizing productivity of developers, 261–262

- overview of, 255
- planning iterations, 259
- project managers and, 256
- quality measures, 260–261
- setting priorities, 259
- summary, 261–262

Development server

- deploying application to App Engine, 15
- starting, 9–11
- starting from command line, 15

Digg, 285

Directory

- command-line tools for deploying, 15
- setting up structure of, 33–34

DOM (Document Object Model), 70

Downgrade tasks, 54

Dragging and dropping items, to pages, 74–76

Duplication of data, normalization and, 45–46

Dynamic instances, avoiding cold startups and, 24

Dynamic interactions on server. See AJAX

E

E-mail

- comparing low-level API to JavaMail, 161
- configuring servlet to receive mail, 161–162
- error handling, 167
- how long it takes to send, 167–168
- logging sent mail, 158–159
- modifying mail queues, 174–175
- overhead of cold startups, 168
- overview of, 155
- parameterizing mail body, 158

- performance and quotas, 167
- performance of mail receiver, 168–169
- queuing send mails, 172–173
- reading without using JavaMail API, 165–166
- receiving, 161
- securing servlet that sends e-mail, 158–159
- sending confirmation e-mails, 155–158
- sending confirmation with JavaMail API, 159–161
- storing received mail, 162–164
- summary, 169

Eclipse tools

- alternatives to, 15
- installing plugins, 4–7
- launching application deployment from Eclipse toolbar, 11
- tools for working with App Engine, 3

ECMAScript 5, 111

Effective Java, Second Edition (Bloch), 209

Elements, selecting in CSS3, 90–91

Entities

- creating/maintaining relationships among, 50–52
- multitenancy affecting, 144
- overview of, 47–48
- transactions between related, 52
- when to separate, 49–50

Entity groups

- parent-child relationships and, 52–53
- using sparingly, 141

Environment variables, specifying deployment parameters, 36

Error handling

- e-mail, 167
- error message types in XMPP, 244
- minimizing damage from failures, 264

- registering memcache error handlers, 213
- URL Fetch API, 221–222

ETag

- browser Cache with ETag headers, 206–208
- retrieving ETag values from cache, 213

Event bubbling, 109

Event delegation, 109–110

Event handling, 109

Exception handling. *See* Error handling

Experiment-driven development, 260

Extensible Messaging and Presence Protocol. *See* XMPP (Extensible Messaging and Presence Protocol)

External feeds, in data modeling, 62

External users, in data modeling, 62

F

Facebook

- connecting via Facebook Apps, 282
- OAuth and, 235
- publishing Facebook pages, 281–282
- social bookmarking, 285
- XMPP and, 241

Fail fast, 264

Failures, minimizing damage from, 264

Files/folders

- HTML form for uploading, 148–149
- reading images from, 193–195
- resizing images, 195–197
- retrieving from blobstores, 150–151
- setting up directory structure, 33–34
- storing large. *See* Blobstores

Filters, servlet filters, 144

Fine-grained caching, 209–210

Flexibility, advantages of cloud computing, 17

Flipping images, 198

Fluent API, 175

Folders. See **Files/folders**

Forms

fetching and storing in datastore,
130–131

for fetching data, 132

improvements in HTML5, 76–77

posting using low-level URL Fetch
API, 223–224

for uploading files, 148–149

Frameworks

AJAX without, 113

choosing for application design, 58–59

minimizing use of, 99

working with/working without, 59–60,
112

**FreeMarker, for generating text from servlets,
60**

Frontend, deployment request received by, 31

Functional languages, 106

Functionality, improving

choosing over appearance, 265

experiment-driven development and,
260

making gradual changes, 260

overview of, 258

planning iterations, 259

setting priorities, 259

Functions, assigning to variables, 106

G

Geolocation API, 77–78

GET method

calling tasks, 175

fetching form data, 130–131

handling page flow, 65

sending and reading data returned from
server, 215

getAll method, for cache values, 213

getMessageType method, XMPP, 244

GIF format, Image API reading, 187

**Global variables, avoiding use of,
110–112**

Goals, in development process, 256–257

Google

AdWords, 284

BigTable, 45

blogging and, 280

dynamically starting/stopping
application servers based on requests
received, 31–32

Gruyere, 239

resource pricing table, 18

search engine optimization, 285

Talk, 241–243

tools for working with App Engine,
3–4

Website Optimizer, 260

Google Accounts API

authenticating users, 229–232

displaying information of logged in
user, 229–231

displaying login/logout URLs, 231–
232

Google Analytics

experiment-driven development and,
260

measuring conversion rate when using
AdWords, 284

measuring user response to interface,
273

optimizing usability, 265

scripting with, 273–275

tracking user actions, 263

Google Apps Marketplace

payment process and, 286–287

reaching your audience, 282–283

Google Web Toolkit (GWT)

- installing Eclipse plugins and, 6
- starting new App Engine project and, 8

Grails framework, for application design, 58**Graphical effects, selecting in CSS3, 92–94****Groupchat, XMPP message types, 244****GWT (Google Web Toolkit)**

- installing Eclipse plugins and, 6
- starting new App Engine project and, 8

H

Headline, XMPP message types, 244**High replication mode, 46****HTML**

- AJAX communicating with server using, 118–119
- cleaning up, 102–106
- displaying availability capabilities with, 268–269
- form for uploading a file, 148–149
- interpreting results of URL Fetch, 216
- links to posts, 134–135
- presenting posts in, 136
- presenting stored data, 132–133
- reducing dependence on JavaScript by enhancing, 106–108
- setting up for AJAX examples, 114
- storing data asynchronously, 137

HTML5

- adopting, xxii–xxiii
- basic elements, 70–72
- canvas element, 72–74
- dragging and dropping items to pages, 74–76
- form elements in, 76–77
- geolocation options for mobile devices, 77–78

introduction to, 69–70

page load performance in, 26

querying structured data, 81–83

storing data across sessions, 78–80

storing session data, 80–81

summary, 81–83

HTTP

cron reading HTTP headers, 182–185

POST request, 161, 168–169

posting form data using, 223–224

requests, 32

response codes, 179

scalability of, 206

status codes, 179, 222

task queuing and, 180–182

HTTPS

enforcing secure protocols, 238

POST requests, 223–224

securing server access, 227

I

IDEA, 15**IDEs (integrated development environments), 3****IDs (identifiers)**

- creating relationships among entities, 50
- ID attributes in CSS3, 86–88

Image API

composite images, 198–201

creating thumbnails, 195–197

cropping images, 197

overview of, 187

reading images from datastore, 191–193

reading images from resource file, 193–195

reading images from user input, 187–190

rotating and flipping images, 198

- summary, 201
- writing images to datastore, 190–191
- writing images to user output, 193

Images, drawing with canvas element, 72–74

IMAP servers

- App Engine not allowed to connect to, 168
- HTTP POST request as alternative to, 161

increment method, for working with values, 213–214

Incremental development, 259

Indexes

- costs vs. other search mechanisms, 49
- creating, 53–54
- search engine optimization, 285
- specifying datastore indexes, 38

Input

- HTML5 form elements, 76–77
- validating, 239

Instance starts/stops, 24–25

Integrated development environments (IDEs), 3

IntelliJ IDEA, 15

Interactions (server)

- dynamic. *See* AJAX
- static. *See* JavaScript

Interactive web applications, long polls and, 120

International Standards Organization (ISO), 260

Internet

- current developments, xx
- historic trends, xix
- optimizing development process for, 255–256

Internet Explorer

- AJAX frameworks in supporting, 122
- canvas element in, 74

- pros/cons of upgrading to HTML5, 69–70

IP addresses, blacklisting IP ranges, 38–39

ISO (International Standards Organization), 260

Iterations, in development process, 259

J

Jabber. *See* XMPP (Extensible Messaging and Presence Protocol)

Jabber ID (JID), 242

JAR (Java archive) files

- deploying with application, 14–15
- images in, 193
- impact on performance, 21–22
- third-party, 18

Java

- archive files. *See* JAR (Java archive) files
- profiling tools, 271

Java Data Objects (JDO)

- datastores and, 129
- minimizing use of libraries, 34

Java Persistence (JPA)

- datastores and, 129
- minimizing use of libraries, 34

Java Runtime Environment (JRE), 215

Java Server Pages (JSP)

- for generating text from servlets, 60
- handlers for task requests, 173

Java Specification Request (JSR)

- JCache API, 214
- minimizing use of libraries, 34

Java Virtual Machine (JVM) languages, 261, xxiii

JavaDoc, 258

JavaMail API

- comparing with low-level API, 161, 167

- overhead of cold startups, 168
- reading mail without using, 165–166
- sending confirmation emails, 159–161
- storing received mail, 163–164

JavaScript

- AJAX as alternative to, 65
- avoiding global variables, 110–112
- cleaning up HTML, 102–106
- cooperation with CSS, 88
- event delegation for optimized performance, 109–110
- HTML5 and, 70
- improving page load performance, 26
- JavaScript 1.8.5, 111
- reducing dependence on, 106–108
- setting up example in, 99–102
- summary, 112

JavaScript Object Notation (JSON)

- communicating with server using, 116–118
- interpreting results of URL Fetch, 216

JCache API. See JSR 107 JCache API**JDBC, 210****JDO (Java Data Objects)**

- datastores and, 129
- minimizing use of libraries, 34

JID (Jabber ID), 242**Joins, aggregating data without using, 46****JPA (Java Persistence)**

- datastores and, 129
- minimizing use of libraries, 34

JPEG

- compared with PNG, 193
- formats read by Image API, 187

jQuery

- building modules and, 112
- removing class with, 88

JRE (Java Runtime Environment), 215**JRuby code, 173****JSON (JavaScript Object Notation)**

- communicating with server using, 116–118
- interpreting results of URL Fetch, 216

JSP (Java Server Pages)

- for generating text from servlets, 60
- handlers for task requests, 173

JSR 107 JCache API, 214**JSR (Java Specification Request), 34****JVM (Java Virtual Machine) languages, 261, xxiii**

K

Key values, creating relationships among entities, 50**Keywords, specifying in data modeling, 62–63**

L

Large files, storing. See Blobstores**Last-Modified values, retrieving from cache, 213****Layout. See CSS3 (Cascading Style Sheets)****Libraries**

- designing applications, 60–61
- measuring cost of class loading, 18–19
- minimizing use of, 99
- setting up directory structure, 34
- timing a servlet that contains a library, 19–20
- timing a servlet that does not contain a library, 21–22

Licenses, software, 6–7**Links, presenting HTML links to posts, 134–135****Load time**

- improving page load performance, 26
- minimizing articles on page load, 108
- writing JavaScript code and, 103

Local storage API, 78–80

Logs/logging

- configuring log levels, 39
- sent mail, 158–159
- unexpected behavior, 269–271

Long polls, AJAX requests and, 120

Low-level API

- comparing with JavaMail, 161, 167
- overhead of cold startups, 168
- processing data asynchronously, 136
- processing data synchronously, 129–130
- sending confirmation e-mails, 155–158

Low-level memory cache (memcache) API

- caching query results, 210
- caching string values, 204–206
- clearing cache, 212–213
- fine-grained cache, 209
- implementing caching strategy, 206
- implementing `Serializable`, 209–210
- invalidating cache items, 211–212
- maintaining a cache, 210–211
- overview of, 203
- pitfalls of a cache, 203–204
- summary, 214
- using Browser Cache with ETag headers, 206–208
- utility methods, 213–214

Low-level URL Fetch API

- posting data, 223–224
- retrieving data, 217–218

M

Mail service, configuring, 37

Many-to-many relationships, 51

Master/slave replication, 46

Maven build tools, setting up directory structure, 34

Memory cache APIs

- JSR 107 JCache API, 214
- low-level memory cache. *See* Low-level memory cache (memcache) API

Menus, data elements in data modeling, 62–63

Messages, XMPP. *See* XMPP (Extensible Messaging and Presence Protocol)

Metadata, querying for available blobs, 149–150

Micro level, designing data at, 47–48

MIME (Multipurpose Internet Mail Extensions)

- Commons File Upload and, 61
- sending MIME message using JavaMail API, 160

Mobile app stores, 285–286

Mobile devices, geolocation options in HTML5, 77–78

Model-View-Controller (MVC)

- handlers for task requests, 173
- servlets following MVC patterns, 60–61

MoSCoW (must haves, should haves, could haves, won't haves), 259

Multipurpose Internet Mail Extensions (MIME)

- Commons File Upload and, 61
- sending MIME message using JavaMail API, 160

Multitenancy

- configuring, 239
- as introduction to namespaces, 144–146

Multithreaded execution, 224

MVC (Model-View-Controller)

- handlers for task requests, 173
- servlets following MVC patterns, 60–61

N

Names, property, 47–48

Namespaces

- configuring, 239
- multitenancy as, 144–146
- setting up, 144–146

NetBeans, 15

News, reaching your audience, 278–279

Normal message type, XMPP, 244

Normalization of data, 45–46

NoSQL databases, xx–xxi

O

OAuth

- implementing service provider for, 236
- providing third party access, 235–237
- RESTful services and, 226

Object-relational mapping (ORM)

- caching mechanism in, 206
- trend away from, xx–xxi

Objectify API, for working with datastore, 129

onclick handlers, 105–106

One-to-many relationships, 51

Online App Engine, deploying application to, 15

Open source standards

- APIs for working with datastore, 129
- XMPP as, 241

OpenID, authenticating users with, 232–235

ORM (object-relational mapping)

- caching mechanism in, 206
- trend away from, xx–xxi

Overhead, reducing in development process, 256–258

P

Page flow, handling in application design, 65

Page layout. See also CSS3 (Cascading Style Sheets), 93

Page load. See Load time

Parameters

- launching tasks with hardcoded, 173
- parameterizing mail body, 158
- reading task details from, 172

Parent-child relationships

- creating relationships among entities, 50
- reasons for avoiding, 52

Parser generators, ANTLR as, 61

Patterns, memory cache as, 204

Payloads, specifying task, 177–179

Payment service provider (PSP), 286

Payments, handling payment process, 286–287

Performance, of mail receiver, 168–169

Performance optimization

- avoiding cold startups, 24–25
- class loading and, 18–19
- cloud computing and, 17–18
- CSS and, 86
- e-mail and, 167
- event delegation for, 109–110
- JSR 107 JCache API, 214
- low-level API for memory cache. *See* Low-level memory cache (memcache) API
- overview of, 17
- reducing web.xml size to minimum, 22–24
- summary, 27
- timing a servlet that contains a library, 19–20
- timing a servlet that does not contain a library, 21–22

tips/possibilities for, 25–27

Permissions, granting user access to applications, 42–43

Personal data, privacy guidelines for, 240

Personas, knowing your audience, 278

Plain text formats, in e-mail, 155

PNG

compared with JPEG, 193

formats read by Image API, 187

Polling, long polls and interactive web applications, 120

POP3 servers

App Engine not allowed to connect to, 168

HTTP POST request as alternative to, 161

Ports, restricting range, 227

POST method

for handling page flow, 65

for storing data in datastore, 130–131

POST requests, HTTP

e-mail and, 161, 168–169

form data and, 223–224

task queuing and, 180–182

Posts

as data elements in data modeling, 62–63

form data, 223–224

large files to blobstore, 148

minimizing, 100–102

presenting HTML links to, 134–135

presenting in HTML, 136

reading single post asynchronously, 140–141

reading single post from datastore, 135–136

task queuing and, 180–182

Presence notifications, storing XMPP messages in datastore, 249–251

Press releases, reaching your audience, 278

Priorities, in development process, 259

Privacy guidelines, 240

Processes, avoiding redundant, 25–26

Production environment, testing in, 263–264

Productivity, optimizing developer, 261–262

Profiling, in quality assurance, 271–273

Profit making, as goal, 256

Programming

benefits of frameworks for, 59

languages and productivity, 261

Project managers, development process and, 256

Projects, starting new App Engine project, 7–9

Properties

choosing in data modeling, 48–49

indexing using multivalued properties, 54

overview of, 47

types of, 48

Prospects, converting into customers, 286

Protocols, enforcing secure, 238

Proxy servers, caching and, 208

Pseudo-classes, selecting in CSS3, 88–89

Pseudo-elements, selecting in CSS3, 91

PSP (payment service provider), 286

Pull API, 184–185

Q

Quality assurance

availability checks, 265–269

logging unexpected behavior, 269–271

measuring user response to interface, 273–275

overview of, 263

profiling in, 271–273

- summary, 275
- testing in production environment, 263–264
- usability and, 265

Quality measures, in development process, 260–261

Queries

- of asynchronous data, 139–140
- of blobstore content, 149–150
- caching results of, 210
- indexes speeding up, 54
- performing, 53
- of structured data, 81–83
- of synchronous data, 133–134

Queues. See Task Queue API

queues.xml file, 39, 174–175

Quint2, 260

Quotas, e-mail

- managing, 174–175
- performance and, 167

R

- Ranges, blacklisting IP ranges, 38**
- Really Simple Syndication. See RSS (Really Simple Syndication)**
- Receiving e-mail. See E-mail**
- Receiving messages, XMPP, 244**
- RedDit, 285**
- References, entities and, 47–48**
- Relational storage, moving away from in data modeling, 45**
- Relationships, one-to-many and many-to-many, 51**
- Remote API, for uploading bulk data, 151–153**
- Repeating tasks, specifying, 37**
- Repository, installing Eclipse plugins from, 5–6**
- Requirements gathering, in application design, 57–58**

- Resizing images, 195–197**
- resource-files, 35–36**
- Resources, optimizing payment for, 18**
- REST API, 184–185**
- RESTful services, 226**
- ResultSets, caching query results in, 210**
- Revealing Modules pattern, 111**
- Rotating images, 198**
- Rounding edges, in CSS3, 94**
- RSS (Really Simple Syndication)**
 - cron and, 182
 - HTTP status codes and, 222
 - URL Fetch API and, 216
- Run As, starting development server, 9**

S

- Safari browser, 76**
- Sanity checks, in test environment, 264**
- SAX (Simple API for XML) parser, 216**
- Scalability**
 - advantages of cloud computing, 17
 - of HTTP, 206
 - separating entities over multiple machines, 47
 - transactions and, 52
- Scalable vector graphics (SVG), 72**
- Scheduling tasks**
 - cron. *See* Cron
 - specifying repeating tasks, 37
 - Task Queue API, 176, 180
- Schemas, benefits of schemaless data, 47**
- SDK (Software Development Kit)**
 - Google tools for working with App Engine, 3
 - setting up directory structure, 34
- Seam framework, in application design, 58**
- Search engine optimization (SEO), 285**

Searches. *See also* **Queries**

- full text, 62
- indexes speeding up, 54
- indexes vs. other search mechanisms, 49
- performing, 53

Secure Sockets Layer (SSL), 36–37**Security**

- of applications, 36–37
- authenticating users with Google Accounts, 229–232
- authenticating users with OpenID, 232–235
- configuring multitenancy, 239
- of e-mail servlet, 158–159, 162
- enforcing authentication, 238
- enforcing secure protocols, 238
- of personal data, 239
- providing access to third parties using OAuth, 235–237
- securing URLs in web.xml, 237
- URL Fetch API, 226–227
- of URL task space, 174
- of URLs, 40–41, 237
- validating input, 239

Selling applications

- approach to, 277
- blogging, 279–280
- converting prospects into customers, 286
- Facebook, 281–282
- Google AdWords, 284
- Google Apps marketplace, 282–283
- handling payment process, 286–287
- knowing your audience, 277–278
- making news and writing articles, 278–279
- mobile app stores, 285–286
- overview of, 277

- reaching your audience, 278
- search engine optimization, 285
- social bookmarking, 285
- summary, 286–287
- Twitter, 280–281

Sending e-mail. *See* **E-mail****Sending messages, using XMPP.** *See* **XMPP (Extensible Messaging and Presence Protocol)****Sent mail, logging, 158–159****SEO (search engine optimization), 285****Serializable, implementing serializing objects, 209–210****Servers**

- AJAX communicating with using HTML, 118–119
- AJAX communicating with using JSON, 116–118
- AJAX communicating with using XML, 115–116
- configuring Remote API on, 151–152
- sending GET requests and reading data returned from, 215
- setting up new channel from, 120–121

Services

- configuring, 37
- overhead costs of invoking, 32

ServletExceptions, 167**Servlets**

- configuring to receive mail, 161–162
- filters for multitenancy, 144
- generating text from, 60
- reducing web.xml size to minimum, 22–24
- securing e-mail Servlet, 158–159, 162
- storing received mail, 162–164
- timing a servlet that contains a library, 19–20

timing a servlet that does not contain a library, 21–22

URL spaces handled by, 65

viewing available, 10–11

Session storage API, 80–81

Sessions

settings for, 36

storing data across, 78–80

storing session data, 80–81

Simple API for XML (SAX) parser, 216

Simple Mail Transfer Protocol (SMTP)

HTTP requests, 32

sending e-mail and, 160

Simple Object Access Protocol (SOAP), 226

SimpleDS API, 129

Slashdot, 285

SMTP (Simple Mail Transfer Protocol)

HTTP requests, 32

sending e-mail and, 160

SOAP (Simple Object Access Protocol), 226

Social bookmarking, 285

Software developers. *See* Developers

Software development. *See* Development process

Software Development Kit (SDK)

Google tools for working with App Engine, 3

setting up directory structure, 34

Sonar, 260

Spring MVC framework, in application design, 58

Spring Roo framework, in application design, 58

SQL databases

querying structured data, 81–83

replacing with NoSQL databases, xx–xxi

Squid page caching, 206, 208

SSL (Secure Sockets Layer), 36–37

Standard URL Fetch API. *See* URL Fetch API

Startup

avoiding cold startups, 24–25

performance hit at, 18

Stateless applications, 78

static-files, 35–36

Static interactions, on server. *See* JavaScript

Status, checking availability of App Engine services, 266–267

Storage

blobstores. *See* Blobstores

data. *See* Data storage

datastores. *See* Datastores

moving away from relational storage, 45

Strings

caching string values, 204–206

hardcoding, 60

specifying as task payload, 177–178

StringTemplate

libraries required in application design, 61

third-party JARs, 18

timing a servlet that contains a library, 19–20

timing a servlet that does not contain a library, 21–22

tools for generating text from servlets, 60

Structured data, querying, 81–83

Struts framework, in application design, 58

Subscription notifications, XMPP messages, 246–248

Suggestions, Google Apps Marketplace, 283

SVG (scalable vector graphics), 72

Synchronous APIs

processing, 129–130

querying, 133–134

retrieving, 135–136

storing, 130–133

System, specifying system properties, 36

T

Table of contents, in data modeling, 62

Tags, in data modeling, 62–63

Target audience

knowing your audience, 278

of this book, xiii–xiv

Task Queue API

benefits of, 179

calling tasks, 175

configuring, 37

configuring task queues, 174

delaying tasks, 176

managing quotas, 174–175

overview of, 171

postponing long-running tasks, 26

queuing send mails, 172–173

referencing admin user, 131–132

running e-mail in background, 167

scheduling tasks, 176

specifying task payloads, 177–179

summary, 182–185

Task queues

configuring, 39

HTTP requests and, 32

Task scheduling. See Scheduling tasks

TDD (test-driven development)

cutting away unnecessary activities, 258

unit testing in, 257

Template engine, choosing for application design, 60

Templates

HTML template for presenting links to posts, 134–135

HTML template for presenting stored data, 132–133

HTML template for storing data asynchronously, 137

Test-driven development (TDD)

cutting away unnecessary activities, 258

unit testing in, 257

Tests

A/B testing, 260

in production environment, 263–264

TDD (test-driven development), 257–258

Text

full text search, 62

markup with CSS declarations, 93

tools for generating from servlets, 60

Third-party

access using OAuth, 235–237

JAR (Java archive) files, 18

Thread-safe mode, 25

Thumbnails, of large images, 195–197

Time

optimizing productivity of developers, 261–262

scheduling tasks, 176

Timeouts, controlling in URL Fetch API, 219–221

Traffic spikes, handling in cloud computing, 17

Transactions

performing, 52–53

scheduling tasks in, 179

storing data in, 142–144

using sparingly, 141

Tree structure, entities in, 53

Twig Persist API, 129

Twitter, 280–281

U

Unit testing, 257

Update tasks, 54

Upgrades, involving datastores, 54

Uploading bulk data, Remote API for, 151–153

Uploading files

for dynamic deployment, 31–32

HTML form for, 148–149

URL class, Java API, 215

URL Fetch API

consuming web services, 226

controlling timeouts, 219–221

exception handling, 221–222

overview of, 215

posting form data, 223–224

reading and interpreting results, 218

retrieving data, 215–218

retrieving responses asynchronously,
224–226

security considerations, 226–227

sending GET requests and reading data
returned from server, 215

summary, 227

writing results to cache or to datastore,
219

URLConnection class, Java API, 215

URLs

choosing model for in application
design, 64–65

configuring with wild card, 248–249

displaying login/logout URLs, 231

enforcing confidential communication
at URL level, 238

handling page flow, 65

posting large files to blobstore, 148

restricting access at URL level, 238

securing, 40–41, 174, 237

Usability, quality assurance and, 263, 265

User API, 232–235

User interface

measuring user response to, 273–275

presenting with HTML5. *See* HTML5

Users

authenticating with Google Accounts,
229–232

authenticating with OpenID, 232–235

as data elements, 62–63

granting access to applications, 42–43

information of logged-in user, 229–231

measuring response to interface, 273–275

reading images from user input, 187–190

writing images to user output, 193

UserService, 229–231

V

Validating input, 239

Values

incrementing, 213–214

property, 47–48

storing in datastore, 137–138

Variables

assigning functions to, 106

avoiding use of global, 110–112

Varnish, 206

Velocity text generation tool, 60

Version

setting current, 42

specifying deployment parameters, 34

Views, separating from data model, 49–50

W

W3C (World Wide Web Consortium)

HTML5 specifications, 70

XForms standard, 77

WAR (web archive)

- App Engine not accepting, 33
- deploying directory as, 15

Warm-up requests, preloading classes using, 24**Web applications**

- authenticating users with Google Accounts, 229–232
- authenticating users with OpenID, 232–235
- caching mechanisms in, 206
- comparing App Engine to traditional, 18
- enforcing authentication, 238
- enforcing secure protocols, 238
- providing third party access, 235–237
- securing URLs in web.xml, 237
- validating input, 239

Web archive (WAR)

- App Engine not accepting, 33
- deploying directory as, 15

Web browsers

- AJAX support, 122
- coding details and standards and, 104–105
- CSS3 support, 85
- ETag support, 208
- HTML5. *See* HTML5
- improving page load performance, 26
- power of writing code, 99
- pros/cons of upgrading to latest, 69
- rules for specificity calculations for CSS, 86
- saving images to, 195–197
- using Browser Cache with ETag headers, 206–208

/WEB-INF folder

- images in, 193

- logging.properties file, 159
- as resource file, 36

Web pages

- caching, 206
- improving page load performance, 26

Web services, consuming, 226**Weblogs**

- blogging and, 279–280
- designing. *See* Designing applications

web.xml

- configuring URLs with wild card in, 248–249
- listening for e-mail addresses, 161–162
- minimizing size of, 22–24
- restricting access at URL level, 238
- securing e-mail Servlets, 162
- securing URLs, 237
- security-constraints added to, 158–159

Wicket framework, in application design, 58**Wild card, configuring URLs with, 248–249****Wordpress tool, for blogging, 279****World Wide Web Consortium (W3C)**

- HTML5 specifications, 70
- XForms standard, 77

X

XForms standard, 77**XHTML**

- comparing HTML5 with, 71–72
- formatting in e-mail, 155

XML

- AJAX communicating with server using, 115–116
- changing to JSON, 117–118

interpreting results of URL Fetch, 216

XForms standard based on, 77

XMPP (Extensible Messaging and Presence Protocol)

configuring, 37

configuring services in appengine-web.xml, 245

configuring URLs with wild card in web.xml, 248–249

HTTP requests, 32

overview of, 241–242

receiving messages, 244

receiving subscription notifications, 246–248

sending messages, 224

sending messages to Google Talk client, 242–243

storing presence notifications in datastore, 249–251

storing received messages in datastore, 244–246

summary, 251

XSLT transformations, for generating text from servlets, 60

Y

Yahoo! logging in using OpenID, 233–234

This page intentionally left blank