Adriaan de Jonge

Essential App Engine

Building High-Performance Java Apps with Google App Engine

Developer's Library

Essential App Engine

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

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

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

Contents

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.

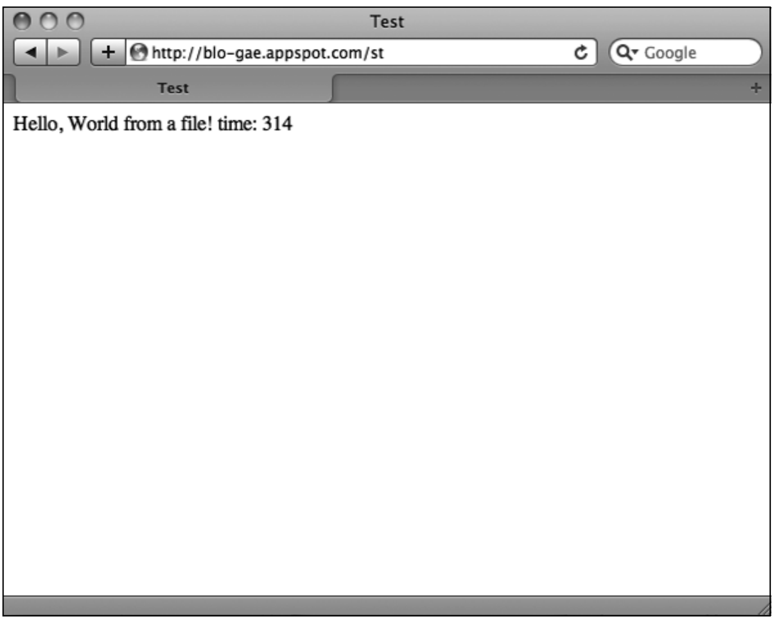

Figure 2.1 Displaying the resulting time in the browser screen with StringTemplate.

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.

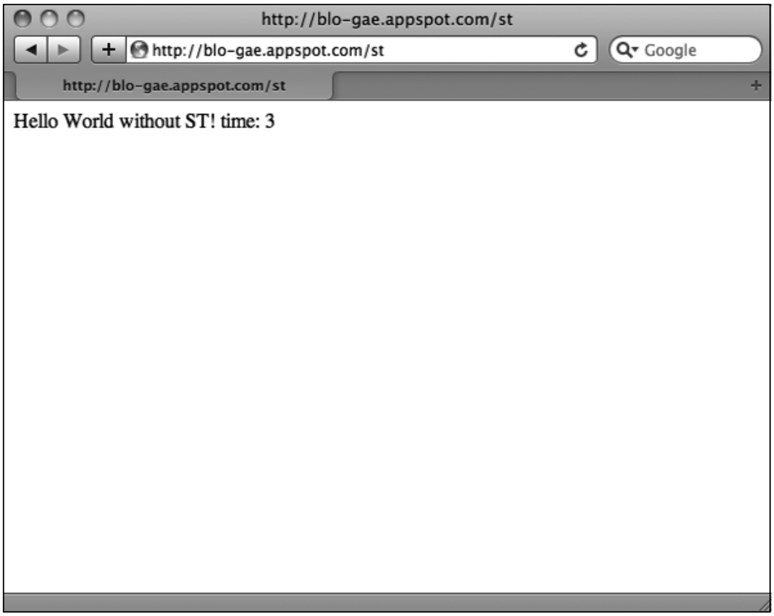

Figure 2.2 Displaying the resulting time in the browser screen without StringTemplate.

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**

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

```

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

```
  <!-- Template -->
  <servlet>
    <servlet-name>StringTemplateServlet</servlet-name>
    <servlet-class>
       com.appspot.template.StringTemplateServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>StringTemplateServlet</servlet-name>
    <url-pattern>/st</url-pattern>
  </servlet-mapping>

</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

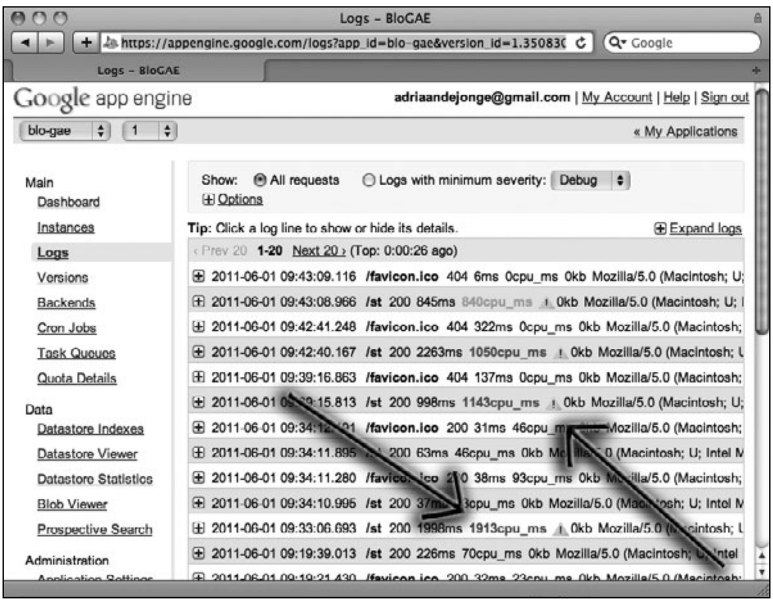

Figure 2.3 Displaying the logged CPU times before and after a web.xml reduction.

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

A

D

E

H

I

J

K

L

M

N

O

P

Q

R

S

U

V

W

X

Y

This page intentionally left blank