ScalaOnGAE Documentation

Release 1.0

Chris Horuk

November 05, 2014

Contents

Contents:

CHAPTER 1

Introduction

This documentation revolves around a technology investigation into the Java implementation of the Google App Engine (GAE). The purpose of this documentation is to help other developers get started using GAE quickly so that they can focus on creating their apps. As the name suggests, much of the programming in this documentation is implemented using the Scala programming language. As such, the detailed instructions presented throughout will be geared specifically towards using Scala with GAE; with the exception of the views, which are written using JSPs.

Another important note is that I am using OS X Mountain Lion. For this reason, some of the instructions will be geared specifically towards Macs, but I will always try to link to some generic downloads, etc. for others.

CHAPTER 2

Installation

To be certain that anyone can easily follow along with this documentation, I will first discuss the installation and setup required. Some of this is required to use Google App Engine (GAE), while some of it is required because of the specific design decisions that I made (i.e. using Scala and Ant).

2.1 Google App Engine Setup

First, I will discuss the setup required to use the Java implementation of GAE.

1. The most obvious part of this setup process is to ensure that you have the latest Java Developer Kit (JDK) installed. I will be using Java 7 throughout, but you can get away with Java 6 if absolutely necessary.
2. The next most obvious step is to download the latest GAE SDK. I am using version 1.7.7 of the GAE Java SDK, but there might be a newer one out if you are reading this in the future (hello from the past!). You'll find plenty of information provided by Google if you are having trouble with this step, but once you download the GAE SDK you should have a top level directory of the form "appengine-java-sdk-x.x.x" where "x.x.x" will be the SDK version number. For clarity, just rename this directory to "appengine-java-sdk" as this is how I will refer to it.
3. This step introduces the first point where you have multiple options to choose from. To facilitate the development process when working with GAE, a project configuration tool of some sort is needed. For those of you who like using IDEs, Eclipse has incredible support for GAE (see Google Plugin for Eclipse). I chose to go a different route and use a command line configuration tool called Apache Ant. You can find some information about how to install Ant on their website, but I will discuss this in some detail later as well.

2.2 Scala Setup

As Scala will be used for much of the programming throughout this documentation, you will of course need to setup Scala. For all of you OS X users out there, this setup should be hilariously easy. Go get Homebrew if you don't have it yet and then simply type *brew install scala* at a Terminal prompt to get the latest stable release. As of this writing, the latest stable release is version 2.10.1, which is what I will be using.

For those of you not using OS X (what are you doing with your life?), you can get the latest stable release of Scala here.

2.3 Ant Setup

Apache Ant is a really useful command line tool that allows you to accomplish the repetitive tasks involved with developing and deploying quickly and efficiently.

1. To use Apache Ant you must have a Java environment installed, but you need this for GAE as well so you should already have it, right?
2. Next, download one of the ant binaries and uncompress the file into a folder where you wish Ant to reside.
3. Once you have found a home for Ant, you need to set an environment variable called ANT_HOME to the directory where Ant resides. You should also have an environment variable called JAVA_HOME and SCALA_HOME at this point as well.

   > **Note:** If you are using OS X, there is a very easy way to set these environment variables that requires administrator privileges. You can set them all in the file */etc/launchd.conf* in the form "setenv ANT_HOME /path/to/apache-ant-x.x.x" on separate lines. After saving this file, simply enter *grep -E "^setenv" /etc/launchd.conf | xargs -t -L 1 launchctl* at the Terminal prompt and restart the Terminal app. Now when you enter the command *export* with no options, you should see your new environment variables listed among others.

4. Finally, you need to add the Ant bin to your PATH.

   > **Note:** On OS X, this can be done by adding "${ANT_HOME}/bin" to the */etc/paths* file.

If you are successful, *ant -version* should return the version of Ant that you have installed. I am using version 1.9.0 because this version is the most recent stable release as of this writing.

CHAPTER 3

Configuring Ant

Google provides a lot of information about how to configure Ant for use with Java, but most of this information does not apply if you wish to use Ant with Scala. This section will explain how to configure Ant for use with Scala and will repeat some of the information from the Google docs that still applies for Scala.

3.1 Initial Configurations

The *build.xml* file is used to configure Ant and a new *build.xml* file should be placed in the top level directory of each project you wish to use it for. There are a number of tasks you can accomplish with Ant, but to use it with GAE there are some initial configurations you need to do.

3.1.1 Import the GAE Ant Macros

There are a number of GAE Ant macros that allow you to execute GAE commands through Ant. In order to use these macros, you must tell Ant where to look for them, which can be accomplished with the following two lines:

```
<property name="sdk.dir" location="../appengine-java-sdk" />
<import file="${sdk.dir}/config/user/ant-macros.xml" />
```

> **Note:** This assumes that you have placed the GAE Java SDK in the directory which contains your project's top-level directory.

3.1.2 Set the Project CLASSPATH

To set the CLASSPATH that Ant will use when executing tasks, you simply need to create a path element with an id of "project.classpath". You typically include any extra JARs that your project needs in the *war/WEB-INF/lib* directory and thus your CLASSPATH will need to include those JARs as well as the GAE SDK JARs.

```
<path id="project.classpath">
    <pathelement path="${build.dir}" />
        <fileset dir="${lib.dir}">
            <include name="**/*.jar" />
        </fileset>
        <fileset dir="${sdk.dir}/lib">
            <include name="shared/**/*.jar" />
```

```
        </fileset>
</path>
```

Note: All of the Ant examples here will use properties defined earlier in the *build.xml* file. Refer to the complete *build.xml* sample files included in the github samples for clarity.

3.2 Creating Targets

If you have worked with Ant before then you already know all about targets. For those that haven't used Ant, targets are, in a nutshell, how you specify the tasks that Ant can execute. You can create them with the <*target name=""*> directive and add dependencies so that certain targets are always executed. Then you can execute them at the command line with the format *ant targetName*, where targetName is the name you specify in the *build.xml* file.

3.2.1 The Scala Init Target

To be able to use Scala commands within Ant, you need an initializer target that will load up all of the Scala commands that Ant can use. In all of the *build.xml* files, this is the target with the name "init". Once this target is created, you can use Scala commands within Ant by making the target that uses the Scala command depend on the "init" target. For clarity, the "init" target is displayed below.

```
<target name="init">
    <path id="build.classpath">
        <pathelement location="${scala-library.jar}" />
        <pathelement location="${build.dir}" />
    </path>
    <taskdef resource="scala/tools/ant/antlib.xml">
        <classpath>
            <pathelement location="${scala-compiler.jar}" />
            <pathelement location="${scala-library.jar}" />
            <pathelement location="${scala-reflect.jar}" />
        </classpath>
    </taskdef>
</target>
```

3.2.2 A Scala Compile Target

One target that clearly must depend on "init" is the "compile" target, as it will need to use the *scalac* command in order to compile the *.scala* files. A sample "compile" target might look like the following:

```
<target name="compile" depends="copyjars, init" description="Compiles Scala source and copies other source files to the WAR.">
    <mkdir dir="${build.dir}" />
    <copy todir="${build.dir}">
        <fileset dir="${src.dir}">
```

```
            <exclude name="**/*.scala" />
        </fileset>
    </copy>
    <scalac srcdir="${src.dir}" destdir="${build.dir}" classpathref="project.classpath" />
</target>
```

Note: The "copyjars" target simply copies all of the JARs that GAE will need when executing your application. Refer to the complete *build.xml* sample files included in the github samples for clarity.

3.2.3 A Target for Starting the Development Server

Here's where those GAE Ant macros that we imported are going to come in handy. A common workflow when developing is to make some changes, ensure that the files all still compile and then finally ensure that they behave as expected when running. You can address this entire workflow with one command at the Terminal prompt using the "runserver" target displayed below, i.e. *ant runserver*. Because it depends on the "compile" target, when you execute the "runserver" target, all actions associated with the "compile" target will be executed first and just like that you can compile all of your Scala code and launch the GAE development server.

```
<target name="runserver" depends="compile" description="Starts the development server.">
    <dev_appserver war="war" port="8888" />
</target>
```

3.2.4 Other Targets

There are a number of other targets that you can define within the *build.xml* file to facilitate development. Take a look at the complete *build.xml* sample files included in the github samples for more ideas.

CHAPTER 4

Getting Acquainted With Google App Engine

Before you begin developing and deploying apps to GAE, there are a few pieces of information that are vital to your success.

4.1 Expected Directory Structure

When deploying your app to GAE, your compiled classes and other resources must be placed within a specific file structure to be properly served by GAE. GAE uses the WAR format, such that any file within the *war* directory in your apps top level directory is considered part of the complete app. A snapshot of the *war* directory might look like the following:

```
war/
  WEB-INF/
    lib/
    classes/
```

Inside of the *war* directory, but outside of the *WEB-INF* directory, you would place any static resources such as image files, as well as any interfaces for your app such as JSP files. Just inside the *WEB-INF* directory, you would place any app configuration files such as the *web.xml* or *appengine-web.xml* files. The *lib* directory is meant to contain the class files of any classes that your app needs while executing, while the *classes* directory contains the class files for your app's classes.

4.2 The Deployment Descriptor (*web.xml*)

To specify your apps routes, require authentication for specific pages, and other web page configuration, you use the *web.xml* file. Every GAE app needs a deployment descriptor as this file is where you map your servlets, JSPs and other files to actual URLs. For a complete description of this file, see the Google Developer docs.

4.3 The App Config Descriptor (*appengine-web.xml*)

For further configurations within your app and to declare your app's registered app ID and version number, you use the *appengine-web.xml* file. One important element to set within this file is the <*threadsafe*> element. By writing <*threadsafe*>*true*</*threadsafe*> in your *appengine-web.xml* file, you are specifying that GAE can use concurrent requests while executing your app. You can also handle configuration such as declaring what files are static (images, CSS, etc.) and what files are resources (JSPs, etc.), configure cache expiration times for static files and much more. For a complete description of this file, see Google's documentation.

4.4 Configure Logging Behavior

All GAE logging is through java.util.logging by default. To configure the logging behavior of your app, you must add the following lines to your *appengine-web.xml* file:

```
<system-properties>
   <property name="java.util.logging.config.file" value="WEB-INF/logging.properties"/>
</system-properties>
```

These lines tell GAE to look in the *war/WEB-INF* directory for a file named *logging.properties* for logging configurations. Inside of this *logging.properties* file you can specify different logging levels:

- FINEST (lowest level)
- FINER
- FINE
- CONFIG
- INFO
- WARNING
- SEVERE (highest level)

A simple *logging.properties* file might look like this:

```
# Set the default logging level for all loggers to WARNING
.level = WARNING
# Specifically set the logging level for the guestbook package to INFO
guestbook.level = INFO
```

CHAPTER 5

Hello, Scala!

Now that you have completed the necessary installations and other setup tasks, you are ready to start coding! This first code example is the obligatory Hello, World! program with a small addition. This addition involves using the Google Users service to allow visitors to sign in with a Google account.

In this code example, you will learn how to:

- Write a simple servlet in Scala to display a webpage.
- Map a Scala servlet to a URL for use by Google App Engine.
- Use the Google Users service to allow users to sign in with a Google account.

5.1 The Configuration Code

First things first, let's set up the deployment descriptor and app config descriptor files so that we can test our app as we go. The deployment descriptor should look like this:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <servlet>
        <servlet-name>helloworld</servlet-name>
        <servlet-class>helloworld.HelloWorldServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>helloworld</servlet-name>
        <url-pattern>/helloworld</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>helloworld</welcome-file>
    </welcome-file-list>
</web-app>
```

This code maps the servlet class *helloworld.HelloWorldServlet* to the servlet name *helloworld* and subsequently maps that servlet name to the */helloworld* URL. Similarly, by placing the servlet's name in the welcome file list, the servlet will be mapped to the root URL as well.

The other file we need is the app config descriptor, which should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application></application>
    <version>1</version>
```

```
    <threadsafe>true</threadsafe>
</appengine-web-app>
```

Note: The *<application>* tag is used to inform GAE of the app's registered ID. Since this app will only be used for testing purposes, it will not have an app ID.

5.2 The Servlet Code

First things first, let's write a simple Scala servlet that will display a webpage with some static content. Create a *HelloWorldServlet.scala* file and add the following code:

```
package helloworld

import javax.servlet.http.{HttpServlet, HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloWorldServlet extends HttpServlet
{
    override def doGet(req : HSReq, resp : HSResp) =
    {
        resp.setContentType("text/plain")
        resp.getWriter().println("Hello World, from Scala")
    }
}
```

This simple servlet will display a webpage with the text "Hello World, from Scala" on it. Deploy your app to the development server and load up the app in your web browser to confirm.

Now for the addition. The Google Users service API provides an interface to allow users to sign in with a Google account, as well as for developers to interact with the current user. The following import statement is needed to use the Google Users service:

```
import com.google.appengine.api.users.{User, UserService => UServ, UserServiceFactory => UServFactory
```

Once you have added this import statement to your *HelloWorldServlet.scala* file, change the *doGet* method to look like the following:

```
    override def doGet(req : HSReq, resp : HSResp) =
{
    val userService = UServFactory.getUserService()
    val user = userService.getCurrentUser()

    if (user != null)
    {
        resp.setContentType("text/plain")
        resp.getWriter().println("Hello, " + user.getNickname() + ", from Scala")
    }
    else
        resp.sendRedirect(userService.createLoginURL(req.getRequestURI()))
}
```

Now, when someone visits the hello world app's webpage, they will either be prompted to sign in to a Google account or a welcome message will be displayed, tailored to their Google account's nickname.

> **Note:** On the development server, the Google Users service simulates the expected behavior by simply allowing you to sign in with any email address you wish without having to enter a password. This allows you to easily test your app when developing.

CHAPTER 6

Storing Data

The chief concern of most web apps, and really any app in general, is how and where to store the app's data. With GAE, you have three different options: a basic schemaless object datastore, a cloud SQL datastore, and an advanced object datastore. The basic schemaless object datastore is where most will start and provides a very scalable option with a choice between using JDO, JPA or a low-level Datastore API. For a more advanced object datastore, Google Cloud Storage allows developers to store their data directly on Google's infrastructure with almost no limit on individual object size and a number of other amazing features. If you need a relational database, the Google Cloud SQL service is exactly what you want: a relational database in the cloud.

In the following sections, I will give examples on how to use each of the three different data storing options with Scala.

6.1 Using the Low-Level Datastore API

The low-level Datastore API gives you access to App Engine's schemaless High Replication Datastore. The Datastore holds objects, referred to as entities, each with one or more properties. Entities are grouped by kind, but two entities of the same kind don't need to have the same properties; the grouping is merely used for the purpose of queries. To retrieve information from the Datastore, you must construct a query with filter parameters to configure how the query's results are sorted. Every query needs one or more Datastore indexes, which are just tables containing entities in an order specified by the index's properties.

The sections that follow are part of a code example displaying how to:

- Generate indexes to support your app's Datastore queries.
- Use JSPs to create your apps interfaces.
- Use the low-level Datastore API to create new entities and execute queries on those entities.

6.1.1 A Note About Indexing

Any interactions you have with your app while it is running on the development server will generate indexes automatically for you as needed. These indexes will be placed inside of a file named *datastore-indexes-auto.xml* inside of a directory named *appengine-generated* in your *war/WEB-INF* directory. You can also specify your own indexes inside of a file named *datastore-indexes.xml* that you must store directly in your *war/WEB-INF* directory. A simple example of this file follows:

```
<datastore-indexes autoGenerate="true">

    <datastore-index kind="Greeting" ancestor="true">
        <property name="date" direction="desc"/>
    </datastore-index>
```

```
</datastore-indexes>
```

The outer *<datastore-indexes>* tag is the wrapper you place around all of the indexes you wish to declare. The *autoGenerate="true"* attribute specifies that you want GAE to take both indexes in this file as well as in the *datastore-indexes-auto.xml* file into account when you are uploading your app. This attribute is important because if a user tries to navigate to a page where a query is executed for which you have no index, the page will fail to load and the index will only then begin to be generated. This process takes some time and user experience will suffer greatly.

If you wish to simulate the behavior of the production environment, you can set *autoGenerate* to "false" in your *datastore-indexes.xml* file and when a query can't be satisfied on your development server, a DatastoreNeedIndexException will be thrown. The exception will also list both the minimal index and the index it would have auto-generated. For a more in-depth discussion of index selection, see this article.

6.1.2 The Deployment Descriptor

The deployment descriptor file for this code example looks like this:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <servlet>
        <servlet-name>sign</servlet-name>
        <servlet-class>guestbook.SignGuestbookServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>sign</servlet-name>
        <url-pattern>/sign</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>guestbook.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Notice that the welcome file is not a servlet, but a JSP file. More on that in the next section.

6.1.3 Creating the User Interface

The user interface for this code example is created in a JSP file to avoid having the servlet code get too messy and to introduce more modularity. The file, named *guestbook.jsp*, looks like this:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="java.util.List" %>
<%@ page import="com.google.appengine.api.users.User" %>
<%@ page import="com.google.appengine.api.users.UserService" %>
<%@ page import="com.google.appengine.api.users.UserServiceFactory" %>
<%@ page import="com.google.appengine.api.datastore.DatastoreServiceFactory" %>
<%@ page import="com.google.appengine.api.datastore.DatastoreService" %>
<%@ page import="com.google.appengine.api.datastore.Query" %>
<%@ page import="com.google.appengine.api.datastore.Entity" %>
<%@ page import="com.google.appengine.api.datastore.FetchOptions" %>
<%@ page import="com.google.appengine.api.datastore.Key" %>
<%@ page import="com.google.appengine.api.datastore.KeyFactory" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<html>
  <head>
    <link type="text/css" rel="stylesheet" href="/stylesheets/main.css" />
```

```
</head>

<body>

  <%
    String guestbookName = request.getParameter("guestbookName");
    if (guestbookName == null) {
      guestbookName = "default";
    }
    pageContext.setAttribute("guestbookName", guestbookName);

    UserService userService = UserServiceFactory.getUserService();
    User user = userService.getCurrentUser();
    if (user != null) {
      pageContext.setAttribute("user", user);
  %>
      <p>Hello, ${fn:escapeXml(user.nickname)}! (You can <a href="<%= userService.createLogoutURL(r
  <%
    } else {
  %>
      <p>Hello! <a href="<%= userService.createLoginURL(request.getRequestURI()) %>">Sign in</a> to
  <%
    }
  %>

  <%
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Key guestbookKey = KeyFactory.createKey("Guestbook", guestbookName);

    // Run an ancestor query to ensure we see the most up-to-date
    // view of the Greetings belonging to the selected Guestbook.
    Query query = new Query("Greeting", guestbookKey).addSort("date", Query.SortDirection.DESCENDIN
    List<Entity> greetings = datastore.prepare(query).asList(FetchOptions.Builder.withLimit(5));
    if (greetings.isEmpty()) {
  %>
      <p>Guestbook '${fn:escapeXml(guestbookName)}' has no messages.</p>
  <%
    } else {
  %>
      <p>Messages in Guestbook '${fn:escapeXml(guestbookName)}'.</p>
  <%
      for (Entity greeting : greetings) {
        pageContext.setAttribute("greeting_content", greeting.getProperty("content"));
        if (greeting.getProperty("user") == null) {
  %>
          <p>An anonymous person wrote:</p>
  <%
        } else {
          pageContext.setAttribute("greeting_user", greeting.getProperty("user"));
  %>
          <p><b>${fn:escapeXml(greeting_user.nickname)}</b> wrote:</p>
  <%
        }
  %>
        <blockquote>${fn:escapeXml(greeting_content)}</blockquote>
  <%
      }
    }
```

```
    %>

    <form action="/sign" method="post">
      <div><textarea name="content" rows="3" cols="60"></textarea></div>
      <div><input type="submit" value="Post Greeting" /></div>
      <input type="hidden" name="guestbookName" value="${fn:escapeXml(guestbookName)}"/>
    </form>

    <form action="/guestbook.jsp" method="get">
      <div><input type="text" name="guestbookName" value="${fn:escapeXml(guestbookName)}"/></div>
        <div><input type="submit" value="Switch Guestbook"/></div>
    </form>

  </body>
</html>
```

The Datastore query executed to retrieve all of the entries in a guestbook requires a Datastore index that you have already seen in the indexing section. The query is executed with the call to prepare() and the result is filtered to only include the 5 most recent entries, which are then displayed on the webpage.

6.1.4 Creating New Guestbook Entries

The sign servlet is where new guestbook entries are created. The code for the sign servlet looks like this:

```
package guestbook

import java.util.logging.Logger
import java.util.Date
import javax.servlet.http.{HttpServlet, HttpServletRequest => HSReq, HttpServletResponse => HSResp}
import com.google.appengine.api.users.{User, UserService => UServ, UserServiceFactory => UServFactory
import com.google.appengine.api.datastore.{DatastoreService, DatastoreServiceFactory => DSFactory, En

class SignGuestbookServlet extends HttpServlet
{
    val log = Logger.getLogger("SignGuestbookServlet")

    override def doPost( req : HSReq, resp : HSResp)
    {
      val userService = UServFactory.getUserService()
      val user = userService.getCurrentUser()

      val guestbookName = req.getParameter("guestbookName")
      val guestbookKey = KeyFactory.createKey("Guestbook",guestbookName)
      val content = req.getParameter("content")
      val date = new Date()

      val greeting = new Entity("Greeting",guestbookKey)
      greeting.setProperty("user",user)
      greeting.setProperty("date",date)
      greeting.setProperty("content",content)

      val datastore = DSFactory.getDatastoreService()
      datastore.put(greeting)
                log.info("Just created new entry in guestbook: " + content + "\nfrom user: " + user)

      resp.sendRedirect("/guestbook.jsp?guestbookName=" + guestbookName)
    }
```

```
}
```

Here you can see a new entity, a "Greeting" entity, is created using the key specific to the name of the guestbook that the user posted in. The name of the user, the current date and time, and the content of the post are all stored with the entity and a message is logged to the INFO level displaying the content of the message and the name of the user that posted it.

6.2 Using the Google Cloud SQL API

Google Cloud SQL is a service that lets you locate your MySQL Databases in Google's cloud. GAE support of Google Cloud SQL is currently experimental, but you can currently try it out for free until next month (June 2013). If you need a relational database for your GAE app, Google Cloud SQL is your answer. There is one key difference to note from the other database options when you are using the Google Cloud SQL service: you must have your own local MySQL instance configured in order to mirror the Google Cloud SQL service during development.

To set up your app to use the Google Cloud SQL service, you need to follow the instructions in the first two sections of this Google documentation. You also need to make sure you have downloaded the mysql-connector-java.jar and placed it inside of your *appengine-java-sdk/lib/impl* directory.

6.2.1 Modifying the Ant Build File

Before we look at the Google Cloud SQL API in use, we need to make a slight modification to the "runserver" target defined in our *buid.xml* file. In order to be able to use Google Cloud SQL services on the development server, we need to pass a few extra command line arguments to the call to launch the development server. The target now looks like this:

```
     <target name="runserver" depends="compile" description="Starts the development server.">
    <dev_appserver war="war" port="8888">
                                    <options>
                                            <arg value="--jvm_flag=-Drdbms.server=local" />
                                            <arg value="--jvm_flag=-Drdbms.driver=com.mysql.jdbc.Dri
                                            <arg value="--jvm_flag=-Drdbms.url=jdbc:mysql://localhos
                                    </options>
                            </dev_appserver>
</target>
```

These three new flags tell the development server to use the local MySQL instance inside of the database named "yourdatabase" with the root user. If you wanted to log in with an account other than the root user, simply replace the string after the question mark with *username=userName&password=password*, replacing *userName* with your desired user name and *password* with the password for that user.

6.2.2 Creating the User Interface

The user interface will once again be created using JSPs for all the same reasons as before, however this time the app itself has changed slightly: users can no longer switch between different guestbooks while posting messages. The main reason behind this change is due to the use of a relational database and the fact that we can no longer create new "groupings" or database tables on the fly. The code looks like this:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="java.util.List" %>
<%@ page import="java.sql.*" %>
<%@ page import="com.google.appengine.api.rdbms.AppEngineDriver" %>
<%@ page import="com.google.appengine.api.users.User" %>
```

```
<%@ page import="com.google.appengine.api.users.UserService" %>
<%@ page import="com.google.appengine.api.users.UserServiceFactory" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<html>
        <head>
        <link type="text/css" rel="stylesheet" href="/stylesheets/main.css" />
        </head>
  <body>

  <%
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
        if (user != null) {
        pageContext.setAttribute("user", user);
  %>
      <p>Hello, ${fn:escapeXml(user.nickname)}! (You can <a href="<%= userService.createLogoutURL(req
  <%
        } else {
  %>
      <p>Hello! <a href="<%= userService.createLoginURL(request.getRequestURI()) %>">Sign in</a> to
  <%
    }
  %>

<%
                Connection connection = null;
                connection = DriverManager.getConnection("jdbc:google:rdbms://instance_name/guestbook
                ResultSet resultSet = connection.createStatement().executeQuery("SELECT guestName, co
%>

                <table style="border: 1px solid black">
                        <tbody>
                                <tr>
                                        <th width="35%" style="background-color: #CCFFCC; margin: 5px
                                        <th style="background-color: #CCFFCC; margin: 5px">Message</t
                                        <th style="background-color: #CCFFCC; margin: 5px">ID</th>
                                </tr>
        <%
                                while (resultSet.next())
                                {
                                    String guestName = resultSet.getString("guestName");
                                    String content = resultSet.getString("content");
                                    int id = resultSet.getInt("entryID");
        %>

                                <tr>
                                        <td><%= guestName %></td>
                                        <td><%= content %></td>
                                        <td><%= id %></td>
                                </tr>

        <%
                                }
                                connection.close();
        %>

                        </tbody>
```

```
                </table>
                <br />
                No more messages!
                <p><strong>Sign the guestbook!</strong></p>
                <form action="/sign" method="post">
                    <div>Message:
                    <br /><textarea name="content" rows="3" cols="60"></textarea>
                    </div>
                    <div><input type="submit" value="Post Greeting" /></div>
                    <input type="hidden" name="guestbookName" />
                </form>
  </body>
</html>
```

The important part of this code is where a new *Connection* object is created. At the call to *DriverManager*'s *getConnection()* method, you would need to replace "instance_name" with the name of your Google Cloud SQL instance and "guestbook" with the name of the database you wish to establish a connection to. A SQL query is then executed to retrieve all of the entries in the entries table of the guestbook database before they are displayed.

6.2.3 Creating New Guestbook Entries

The sign servlet is where new guestbook entries are created. The code for the sign servlet looks like this:

```
package guestbook

import com.google.appengine.api.rdbms.AppEngineDriver
import java.util.logging.Logger
import java.util.Date
import javax.servlet.http.{HttpServlet, HttpServletRequest => HSReq, HttpServletResponse => HSResp}
import com.google.appengine.api.users.{User, UserService => UServ, UserServiceFactory => UServFactory
import java.sql._

class SignGuestbookServlet extends HttpServlet
{
    val log = Logger.getLogger("SignGuestbookServlet")

    override def doPost( req : HSReq, resp : HSResp)
    {
                val out = resp.getWriter()
                var connection:Connection = null
                val userService = UServFactory.getUserService()
                val user = userService.getCurrentUser()
           try {
                DriverManager.registerDriver(new AppEngineDriver())
              connection = DriverManager.getConnection("jdbc:google:rdbms://instance_name/guestbook"
              val content = req.getParameter("content")
              if (content == "")
                out.println("<html><head><link type='text/css' rel='stylesheet' href='/stylesheets/ma
                       else
                       {
                val statement ="INSERT INTO entries (guestName, content) VALUES( ? , ? )"
                val preparedStatement = connection.prepareStatement(statement)
                preparedStatement.setString(1, if (req.getUserPrincipal() != null) req.getUserPrincip
                preparedStatement.setString(2, content)
                var success = 2
                success = preparedStatement.executeUpdate()
                if (success == 1)
```

```
                                out.println("<html><head><link type='text/css' rel='stylesheet' href=
                                else if (success == 0)
                                out.println("<html><head><link type='text/css' rel='stylesheet' href=
            }
        }
             catch
             {
             case e:SQLException => e.printStackTrace()
             }
             finally
             {
            if (connection != null)
             try
                                    connection.close()
               catch
                              {
                     case ignore:SQLException => ()
                              }
        }
             resp.setHeader("Refresh","3; url=/guestbook.jsp")
    }
}
```

Once again, we need to create a *Connection* object and use a *DriverManager* to help initiate the connection. We then create a prepared statement and place the message contents, along with a user name if the poster was signed in, and update the database with this new entry. The last line sets the header on the response to refresh the browser window 3 seconds after the method completes with the *guestbook.jsp* interface.

6.3 Using the Google Cloud Storage API

The Google Cloud Storage service is very similar to the basic Datastore only with less limits placed due to the higher billing costs for using the service. The Google Cloud Storage service is also currently experimental with GAE, but there are a number of features that are not offered by the other datastore options. These include access control lists, OAuth 2.0 authentication and authorization, the ability to resume upload operations if they're interrupted, and a RESTful API among others. One important fact is that all objects created with the Google Cloud Storage service are immutable. To modify an existing object, you must overwrite it with a new object containing your changes.

Before you can begin using the Google Cloud Storage APIs, you need to activate the Google Cloud Storage service for your app and enable billing. The detailed steps can be found in the prerequisites section of this document. Once you have activated the service, you are ready to start using the Google Cloud Storage APIs.

6.3.1 Creating the User Interface

As with the other two code examples, the user interface will be created using JSPs. The code looks like this:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="java.util.List" %>
<%@ page import="com.google.appengine.api.users.User" %>
<%@ page import="com.google.appengine.api.users.UserService" %>
<%@ page import="com.google.appengine.api.users.UserServiceFactory" %>
<%@ page import="com.google.appengine.api.files.AppEngineFile" %>
<%@ page import="com.google.appengine.api.files.FileReadChannel" %>
<%@ page import="com.google.appengine.api.files.FileService" %>
<%@ page import="com.google.appengine.api.files.FileServiceFactory" %>
```

```
<%@ page import="com.google.appengine.api.files.FileWriteChannel" %>
<%@ page import="com.google.appengine.api.files.GSFileOptions.GSFileOptionsBuilder" %>
<%@ page import="java.net.URL" %>
<%@ page import="com.google.appengine.api.urlfetch.HTTPRequest" %>
<%@ page import="com.google.appengine.api.urlfetch.HTTPResponse" %>
<%@ page import="com.google.appengine.api.urlfetch.HTTPMethod" %>
<%@ page import="com.google.appengine.api.urlfetch.URLFetchService" %>
<%@ page import="com.google.appengine.api.urlfetch.URLFetchServiceFactory" %>
<%@ page import="org.w3c.dom.Document" %>
<%@ page import="org.w3c.dom.*" %>
<%@ page import="javax.xml.parsers.DocumentBuilderFactory" %>
<%@ page import="javax.xml.parsers.DocumentBuilder" %>
<%@ page import="java.io.BufferedReader"%>
<%@ page import="java.nio.channels.Channels" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<html>
  <head>
    <link type="text/css" rel="stylesheet" href="/stylesheets/main.css" />
  </head>

  <body>

    <%
                String BUCKETNAME = "YOUR_BUCKET_NAME";
      String guestbookName = request.getParameter("guestbookName");
      if (guestbookName == null) {
        guestbookName = "default";
      }
      pageContext.setAttribute("guestbookName", guestbookName);

      UserService userService = UserServiceFactory.getUserService();
      User user = userService.getCurrentUser();
      if (user != null) {
        pageContext.setAttribute("user", user);
    %>
        <p>Hello, ${fn:escapeXml(user.nickname)}! (You can <a href="<%= userService.createLogoutURL(r
    <%
      } else {
    %>
        <p>Hello! <a href="<%= userService.createLoginURL(request.getRequestURI()) %>">Sign in</a> to
    <%
      }
    %>

    <%

                          URL url = new URL("http://" + BUCKETNAME + ".storage.googleapis.com");

                          HTTPRequest bucketListRequest = new HTTPRequest(url, HTTPMethod.GET);
                          URLFetchService service = URLFetchServiceFactory.getURLFetchService();
                          HTTPResponse bucketLisResponse = service.fetch(bucketListRequest);
                          String content = new String(bucketLisResponse.getContent(), "UTF-8");

                          DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance
                          DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
                          Document doc = docBuilder.parse(content);
```

```
                        // normalize text representation
                        doc.getDocumentElement().normalize();
                        NodeList listOfEntries = doc.getElementsByTagName("Contents");
                        if (listOfEntries.getLength() == 0)
                        {
                %>
                <p>Guestbook '${fn:escapeXml(guestbookName)}' has no messages.</p>
                <%
                        }
                        else
                        {
                %>
                <p>Messages in Guestbook '${fn:escapeXml(guestbookName)}'.</p>
                <%
                                for (int i=0; i<listOfEntries.getLength(); i++)
                                {
                                        Node entryNode = listOfEntries.item(i);
                                        if (entryNode.getNodeType() == Node.ELEMENT_NODE)
                                        {
                                                Element entryElement = (Element) entryNode;
                                                String entryMessageKey = entryElement.getElementsByTa
                                                String fileName = "/gs/" + BUCKETNAME + "/" + entryMe
                                                AppEngineFile readableFile = new AppEngineFile(fileNa
                                                FileService fileService = FileServiceFactory.getFileS
                                                FileReadChannel readChannel = fileService.openReadCha
                                                BufferedReader reader = new BufferedReader(Channels.n
                                                String line = reader.readLine();
                                                pageContext.setAttribute("greeting_content", line);
                %>
                <blockquote>${fn:escapeXml(greeting_content)}</blockquote>
                <%
                                        }
                                }
                        }

    %>

    <form action="/sign" method="post">
      <div><textarea name="content" rows="3" cols="60"></textarea></div>
      <div><input type="submit" value="Post Greeting" /></div>
      <input type="hidden" name="guestbookName" value="${fn:escapeXml(guestbookName)}"/>
    </form>

    <form action="/guestbook.jsp" method="get">
      <div><input type="text" name="guestbookName" value="${fn:escapeXml(guestbookName)}"/></div>
        <div><input type="submit" value="Switch Guestbook"/></div>
    </form>

  </body>
</html>
```

There is a lot going on in the interface code, so let's break it down into sections. The first section being discussed is the section we are familiar with from the other code examples:

```
     <%
                String BUCKETNAME = "YOUR_BUCKET_NAME";
  String guestbookName = request.getParameter("guestbookName");
  if (guestbookName == null) {
```

```
    guestbookName = "default";
  }
  pageContext.setAttribute("guestbookName", guestbookName);

  UserService userService = UserServiceFactory.getUserService();
  User user = userService.getCurrentUser();
  if (user != null) {
    pageContext.setAttribute("user", user);
%>
    <p>Hello, ${fn:escapeXml(user.nickname)}! (You can <a href="<%= userService.createLogoutURL(reque
<%
  } else {
%>
    <p>Hello! <a href="<%= userService.createLoginURL(request.getRequestURI()) %>">Sign in</a> to in
<%
  }
%>
```

This section of the code is merely checking to see if the user is logged in to a Google Account and displaying the appropriate login or logout link. The one addition to this section is the *BUCKETNAME* variable. This variable should be set to the name of the bucket that you created in the prerequisites section of this document.

The next section of the code to examine deals with fetching all of the objects in the bucket specified by the *BUCKETNAME* variable:

```
URL url = new URL("http://" + BUCKETNAME + ".storage.googleapis.com");

HTTPRequest bucketListRequest = new HTTPRequest(url, HTTPMethod.GET);
URLFetchService service = URLFetchServiceFactory.getURLFetchService();
HTTPResponse bucketLisResponse = service.fetch(bucketListRequest);
String content = new String(bucketLisResponse.getContent(), "UTF-8");
```

This section of the code is displaying a few different techniques and services. First off, the GAE URL Fetch service APIs are being used to send a GET request to the bucket we are interested in. This process involves creating an *HTTPRequest* object to represent the GET request, creating a *URLFetchService* object to fetch the request's response and an *HTTPResponse* object to store the response for use. The GET request in question is part of the Google Cloud Storage XML RESTful APIs, which are documented here. As GAE support for Google Cloud Storage is still experimental, there is no set Java method to retrieve the contents of a bucket, which is why we have to resort to this method.

In the next section of the code, we parse the XML response of our GET request:

```
        DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
        Document doc = docBuilder.parse(content);

        // normalize text representation
        doc.getDocumentElement().normalize();
        NodeList listOfEntries = doc.getElementsByTagName("Contents");
        if (listOfEntries.getLength() == 0)
        {
%>
<p>Guestbook '${fn:escapeXml(guestbookName)}' has no messages.</p>
<%
        }
        else
        {
%>
<p>Messages in Guestbook '${fn:escapeXml(guestbookName)}'.</p>
```

The first three lines create a *DocumentBuilder* object that we can use to parse the XML response. We then normalize the text representation so that we can query our newly parsed response content by XML tag names. We get the "Contents" tag and check to see if there are any messages stored within this guestbook's bucket.

The next section of the code represents the logic behind extracting all of the messages from the XML response:

```
<%
                for (int i=0; i<listOfEntries.getLength(); i++)
                {
                        Node entryNode = listOfEntries.item(i);
                        if (entryNode.getNodeType() == Node.ELEMENT_NODE)
                        {
                                Element entryElement = (Element) entryNode;
                                String entryMessageKey = entryElement.getElementsByTagName("Key").ite
                                String fileName = "/gs/" + BUCKETNAME + "/" + entryMessageKey;
                                AppEngineFile readableFile = new AppEngineFile(fileName);
                                FileService fileService = FileServiceFactory.getFileService();
                                FileReadChannel readChannel = fileService.openReadChannel(readableFil
                                BufferedReader reader = new BufferedReader(Channels.newReader(readCha
                                String line = reader.readLine();
                                pageContext.setAttribute("greeting_content", line);
%>
<blockquote>${fn:escapeXml(greeting_content)}</blockquote>
<%
                        }
                }
        }

%>
```

This section is where we use the Google Cloud Storage Java APIs to access each file within the bucket. We first create the file name using the bucket name and the key retrieved from the XML response. We then create an *AppEngineFile* object to represent the file and a *FileReadChannel* object to represent the read channel. Due to the fact that all messages are stored on a single line, we simply retrieve this line from the reader and display it on the page.

The final section of the code represents the same two forms used in the original datastore code sample.

6.3.2 Creating New Guestbook Entries

The sign servlet is where new guestbook entries are created. The code for the sign servlet looks like this:

```
package guestbook

import java.util.logging.Logger
import java.util.Date
import java.io.PrintWriter
import java.nio.channels.Channels
import javax.servlet.http.{HttpServlet, HttpServletRequest => HSReq, HttpServletResponse => HSResp}
import com.google.appengine.api.users.{User, UserService => UServ, UserServiceFactory => UServFactory
import com.google.appengine.api.files.{AppEngineFile, FileReadChannel, FileService, FileServiceFactor
import com.google.appengine.api.files.GSFileOptions.GSFileOptionsBuilder

class SignGuestbookServlet extends HttpServlet
{
    val log = Logger.getLogger("SignGuestbookServlet")
         // You might make this depend on the guest book name for example.
         val BUCKETNAME = "YOUR_BUCKET_NAME"
         // You might make this depend on information specific to the message being posted for exampl
```

```
        val FILENAME = "YOUR_FILE_NAME"

    override def doPost( req : HSReq, resp : HSResp)
    {
      val userService = UServFactory.getUserService()
      val user = userService.getCurrentUser()

      val guestbookName = req.getParameter("guestbookName")
      val content = req.getParameter("content")
      val date = new Date()

                val fileService = FileServiceFactory.getFileService()
                val optionsBuilder = new GSFileOptionsBuilder().setBucket(BUCKETNAME).setKey(FILENAME
                val writableFile = fileService.createNewGSFile(optionsBuilder.build())

                // Lock for writing because we intend to finalize this object.
                val lockForWrite = true
                val writeChannel = fileService.openWriteChannel(writableFile, lockForWrite)
                val out = new PrintWriter(Channels.newWriter(writeChannel, "UTF8"))
                out.println(content)
                out.close()

                // Finalize the object so that it can be read from later.
                writeChannel.closeFinally()
                log.info("Just created new entry in guestbook: " + content + "\nfrom user: " + user)

                resp.sendRedirect("/guestbook.jsp?guestbookName=" + guestbookName)
    }
}
```

The first important thing to note involves the two new values added to the servlet, *BUCKETNAME* and *FILENAME*. While these are represented by constant values, you would likely need to change these for different entries. You might for example base your bucket name on the name of the guest book that you are submitting an entry to. You might base the file name on some unique combination of the message's properties, such as the author's name and the date created. Once you have figured this out, all you need to do is use a *GSFileOptionsBuilder* object to configure your new message and then create a new *GSFile* object to represent it. You then open up a channel for writing and lock it for exclusive access since the message is going to be finalized in this process.

As noted earlier, files stored using the Google Cloud Storage services are immutable once saved. Therefore you need to explicitly finalize the object to state that you are done making changes to it. The servlet ends by redirecting back to the guestbook.

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*