Google App Engine for Java, Part 1: Rev it up!

Building scalable, Java-based killer apps with App Engine for Java

Skill Level: Introductory

Rick Hightower (richardhightower@gmail.com)
CTO
Mammatus Inc.

11 Aug 2009

Remember when Google App Engine was just for *Pythonistas*? Those were some dark days. Google Inc. opened up its cloud-computing platform to Java™ developers in April 2009. In this three-part article series, Java technology author and trainer Rick Hightower gets you started with this reliable, robust, and fun platform for Java-based development. In this article, you'll get an overview of why Google App Engine for Java could be the deployment platform for your next highly scalable killer app, then start using the Google Plugin for Eclipse to build two example apps: one based on Google Web Toolkit (GWT) and one based on the Java Servlet API. You'll learn for yourself what a difference Google App Engine for Java makes, both in building out an application from scratch and in deploying it to the tune of up to five million views. (And that's just the free version.)

An idea is like an itch: you need to scratch it, and when you do it feels better. As software developers, we spend a lot of time thinking up ideas for different kinds of applications. That's fun, right? What's challenging is figuring out how to bring a software product to fruition. It's satisfying to imagine something *and then* create it. The alternative (an unscratched itch) is just frustrating.

One reason many applications never get off of the ground is the need for infrastructure. A well-maintained infrastructure usually involves a team of system administrators, DBAs, and network engineers, which, until recently, was an enterprise mostly for the rich. Even paying a third party to host your application isn't fool-proof: what happens if the app's popularity skyrockets and it suddenly gets a lot

of hits? The so-called *Slashdot effect* can crater a good idea, simply because it's hard to predict load spikes.

But, as we all know, that's changing. The premise of Web services has evolved, and today it's bringing us the means, via cloud computing and its beefier cousin, platform-as-a-service/PAAS, to build, deploy, and distribute applications more easily. Now, when you write the next Twitter and deploy it on a cloud platform, it will scale, baby, scale. Wow, that feels good!

In this three-part article, you'll learn hands-on why cloud computing/PAAS is such an important evolutionary shift for software development, while also getting started with an exciting new platform for Java development: Google App Engine for Java, which is currently available in preview release. I'll begin with an overview of App Engine for Java, including the types of application services it provides. After that you'll dive straight into an application example — the first of two — using the App Engine for Java Google Plugin for Eclipse. The first application example will leverage App Engine for Java's support for the Java Servlet API and the second will leverage its support for GWT. In Part 2, you'll create a small contact-management application using App Engine for Java's support for servlets and GWT, respectively. And in Part 3, you'll use your custom-built application to explore App Engine for Java's Java-based persistence support, which is based on Java Data Objects (JDO) and the Java Persistence API (JPA).

Okay, enough talk: Let's rev!

About Google App Engine for Java

Google (also the maker of some sort of search engine, I believe) first released Google App Engine in April 2008. To the dismay of many Java developers, the initial release was purely the domain of Python programmers — people who think white space should be used for blocks! (I've written a book about Python, so I should know.) Google responded to popular demand by releasing Google App Engine for Java in April 2009.

Google App Engine for Java provides an end-to-end solution for enterprise Java development: a browser-based Ajax GUI for ease of use, Eclipse tool support, and Google App Engine on the back end. Ease of use and tooling are advantages of Google App Engine for Java over other cloud computing solutions.

Application development in App Engine for Java means using Google's resources to store and retrieve Java objects. Data storage is based on BigTable, but with JDO and JPA interfaces that allow you to write code that is not directly tied to BigTable. In fact, Google provides standards-based support for many APIs so that you can write code that is not 100% tied to the App Engine for Java platform.

App Engine for Java relies on the following standard Java APIs:

- `java.net.URL` to fetch services (by communicating with other hosts using the HTTP and HTTPS protocols)
- JavaMail to send mail messages
- A JCache (JSR 107) interface to Memcache to provide fast, temporary distributed storage for caching queries and calculations

Deploying App Engine for Java on WebSphere/DB2
When they announced App Engine for Java, representatives from Google and IBM® deployed the same sample application on DB2®/WebSphere®. IBM is working on providing low-level Google API support for Tivoli® LDAP and DB2 so that applications built for App Engine for Java can also run on the IBM WebSphere/DB2 stack.

In addition, App Engine for Java provides support for the following application services:

- User authentication and authorization
- CRON
- Data import/export
- Access to firewall data

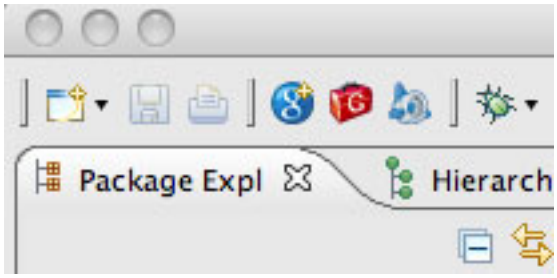
The data import/export is important for moving data from other sources into your App Engine for Java application. This is another way that you are not tied to App Engine for Java. Google's CRON support is based on an internal URL getting hit on a certain schedule, making this a nice service that does not have much tie-in to App Engine for Java. The user authentication and authorization mechanism *is* specific to App Engine for Java, but you could write a `ServletFilter`, aspect, or Spring Security plug-in in order to minimize that tight coupling.

Create your first App Engine for Java application

If you've read this far, then you're ready to start building your first App Engine for Java application. Your first step is to install the Google Plugin for Eclipse for App Engine for Java; with that done, you're good to go.

Open up your Eclipse IDE and you will see three new buttons in your Eclipse IDE next to the Printer button: A *G* in a blue ball, a *G* in a red toolbox, and an App Engine for Java mini-jet plane, as shown in Figure 1:

Figure 1. Shiny new buttons in your Eclipse IDE



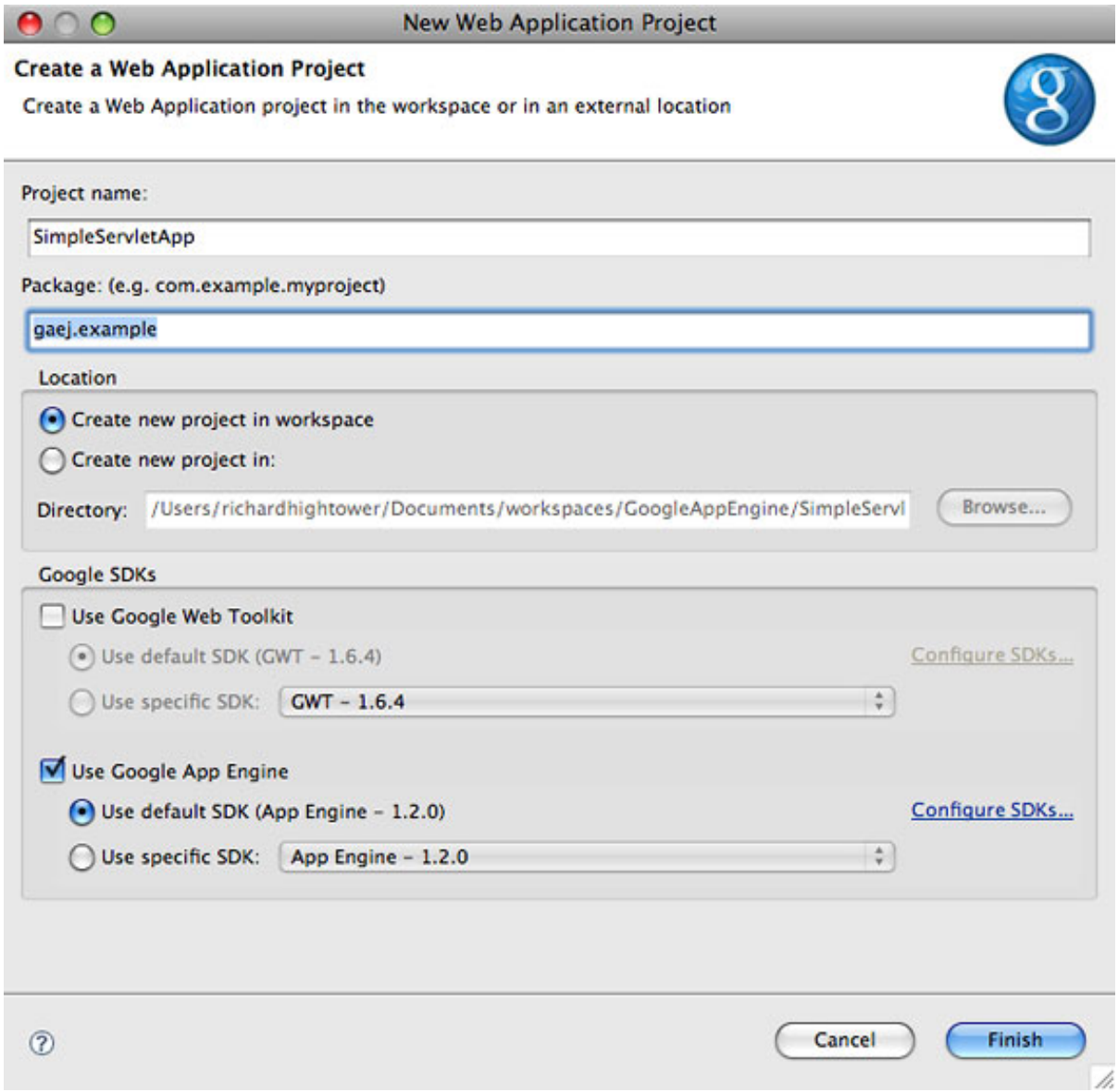
Here's what those buttons do:

- The blue ball lets you access the App Engine for Java project-creation wizard.
- The red toolbox lets you compile a GWT project.
- The mini-jet plane is your key to deploy an App Engine project.

You'll use the project-creation wizard to create two new projects: one based on servlets and the other built using GWT. You'll use the toolbox functionality to compile a GWT project. You'll launch the mini-jet when you're ready to deploy the App Engine project, making it live.

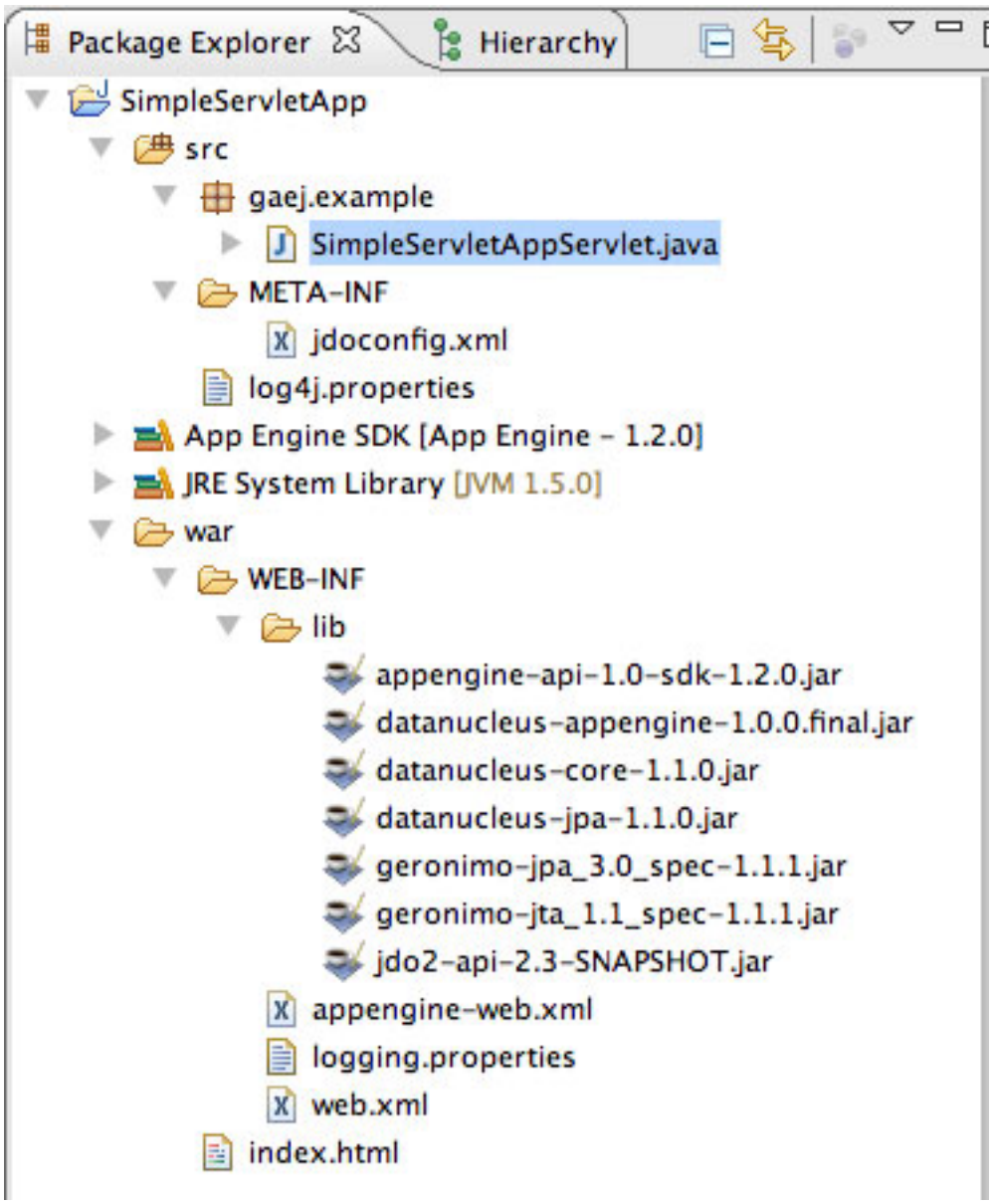
Start now by creating an App Engine for Java project. First, click the blue ball to access the project-creation wizard. Then create an app called SimpleServletApp using the package called gaej.example, as shown in Figure 2:

Figure 2. Starting a new project



Notice how GWT support is unselected for this first simple example. After you complete this step, the project-creation wizard will create a simple servlet-based application featuring a Hello World-type servlet. Figure 3 shows a screenshot of the project:

Figure 3. The SimpleServletApp project



Notice the JAR files that are automatically included for this new, servlet-based project:

- **datanucleus-*.jar**: For accessing the App Engine for Java datastore using standard JDO or the low-level BigTable API
- **appengine-api-sdk.1.2.0.jar**: For using nonstandard App Engine for Java application services like App Engine for Java Security
- **geronimo-*.jar**: For using standard Java APIs like Java Transaction Management API (JTA) and JPA
- **jdo2-api-2.3-SNAPSHOT.jar**: For using the JDO API

You'll learn how to use the persistence APIs from App Engine for Java and some of App Engine for Java's application services starting in Part 2 of this article.

Also notice the file for configuring the runtime container for Google App Engine, called appengine.xml. In this example, appengine.xml is being used to configure the logging.properties file to do logging with App Engine for Java.

First look at an App Engine for Java servlet application

Once you've configured everything in the project-creation wizard, App Engine for Java presents you with the bare bones of a Hello World-style servlet app. Look at the code and then see how to run the application using the App Engine for Java Eclipse tools. The main entry point to this application is the `SimpleServletAppServlet`, as shown in Listing 1:

Listing 1. SimpleServletAppServlet

```
package gaej.example;

import java.io.IOException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class SimpleServletAppServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws IOException {
        resp.setContentType("text/plain");
        resp.getWriter().println("Hello, world");
    }
}
```

The servlet gets mapped under the URI `/simpleservletapp` in the web.xml, as shown in Listing 2:

Listing 2. web.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <servlet>
        <servlet-name>simpleservletapp</servlet-name>
        <servlet-class>gaej.example.SimpleServletAppServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>simpleservletapp</servlet-name>
        <url-pattern>/simpleservletapp</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
```

```
</web-app>
```

The project-creation wizard also provides an index.html file that has a link to the new servlet, as shown in Listing 3:

Listing 3. index.html generated by the project-creation wizard

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- The HTML 4.01 Transitional DOCTYPE declaration-->
<!-- above set at the top of the file will set     -->
<!-- the browser's rendering engine into           -->
<!-- "Quirks Mode". Replacing this declaration     -->
<!-- with a "Standards Mode" doctype is supported, -->
<!-- but may lead to some differences in layout.   -->

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">

    <!--                                           -->
    <!-- Any title is fine                         -->
    <!--                                           -->
    <title>Hello App Engine</title>
  </head>

  <!--                                           -->
  <!-- The body can have arbitrary html, or      -->
  <!-- you can leave the body empty if you want  -->
  <!-- to create a completely dynamic UI.        -->
  <!--                                           -->
  <body>
    <h1>Hello App Engine!</h1>

    <table>
      <tr>
        <td colspan="2" style="font-weight:bold;">Available Servlets:</td>
      </tr>
      <tr>
        <td><a href="simpleservletapp"/>SimpleServletAppServlet</td>
      </tr>
    </table>
  </body>
</html>
```

You now have a simple servlet application built using just some Java APIs. And that is the point: App Engine for Java wraps App Engine functionality using standard Java APIs, enabling App Engine to support the wealth of frameworks available for the Java platform.

What else works with App Engine for Java?

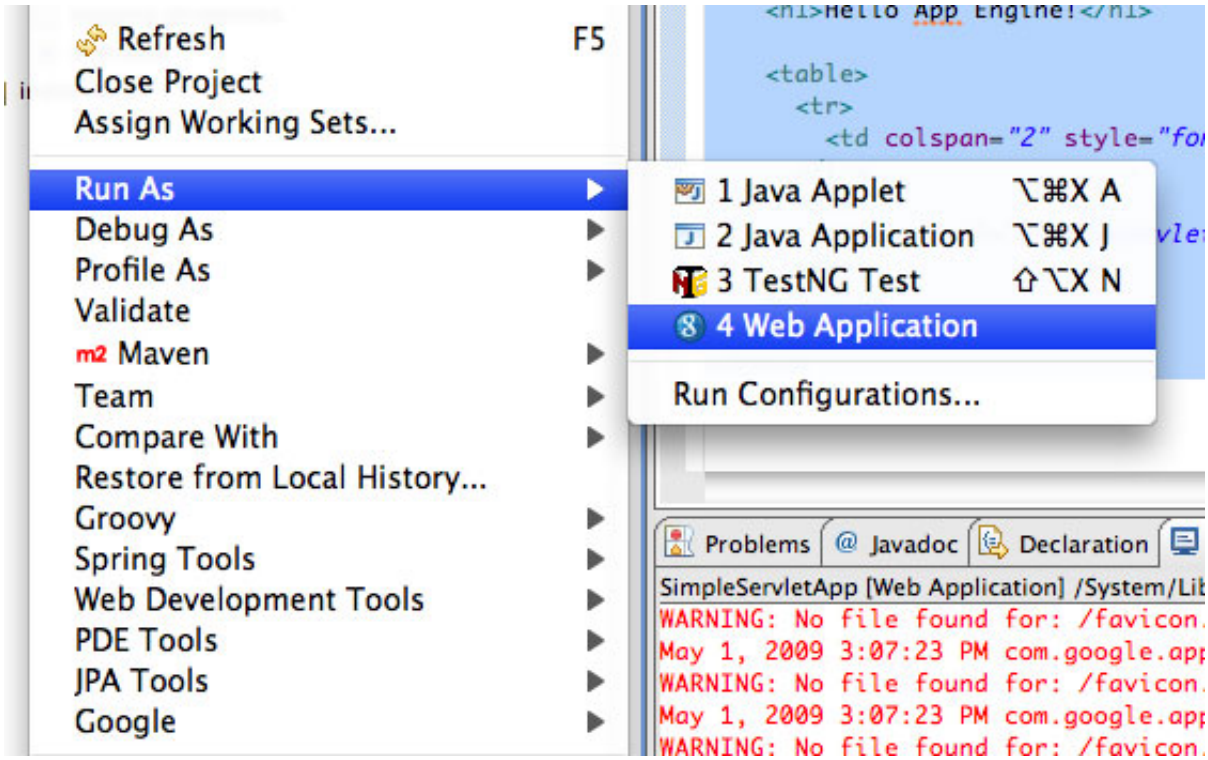
Google maintains a listing of tools and frameworks that work well with App Engine for Java (see Resources). For example, App Engine for Java supports many JVM languages, including BeanShell, Groovy, Scala, JRuby, Jython, and Rhino. Because App Engine for Java supports quite a few of the Java SE and Java EE APIs — like Servlets, JSP, JPA, JavaMail, and the Java API for XML Processing (JAXP) — many existing frameworks work in App Engine for Java out of the box. For example, you can use the Spring

framework, though you'll need some workarounds for Spring ORM. Tapestry, Wicket, DWR, Tiles, SiteMesh, and Grails also work. Struts 2 works with a small patch. Some frameworks that do not work in App Engine for Java are Hibernate and JDBC (no support for relational databases), JMX, Java WebServices, JAX-RPC or JAX-WS, JCA, JNDI, JMS, EJB, and Java RMI.

Deploying the application

To run your servlet-based application with the App Engine for Java Eclipse tools, first right-click the project and select the **Run As** menu, then select "Web Application" with the blue ball beside it, as shown in Figure 4:

Figure 4. Running the App Engine for Java development server



Now you should be able to navigate to http://localhost:8080/simpleservletapp in your browser and see the application with the Hello World message.

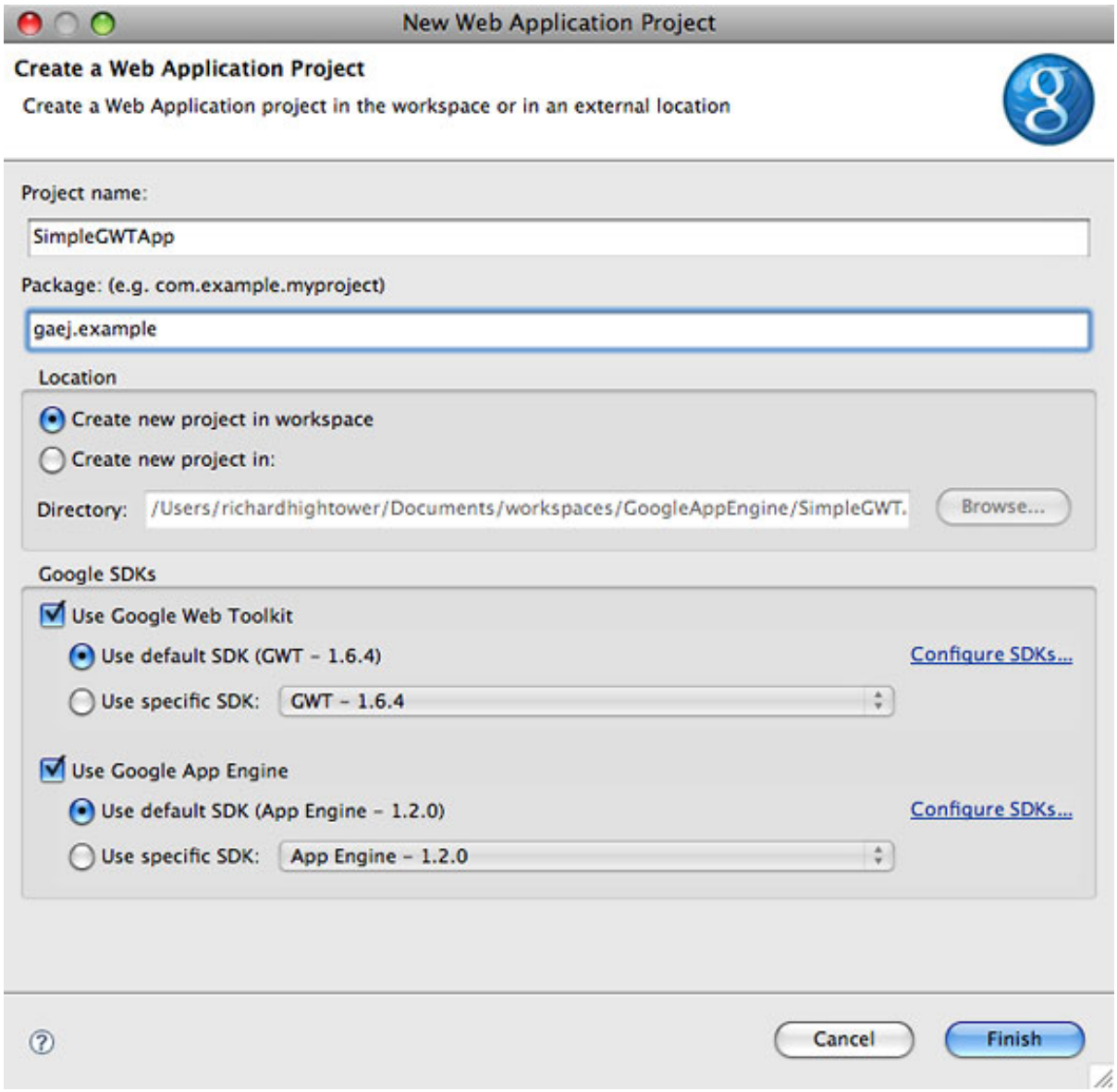
GWT vs. traditional Java Web apps

When working in the GWT, you compile Java code to JavaScript and then run your Web application GUI in the browser. The result is much more like a traditional GUI application than a traditional Java Web application. GWT has a client component that runs in the browser. The client component talks to your Java server-side code through a set of RMI-style Java classes that you write.

Create an App Engine for Java/GWT application

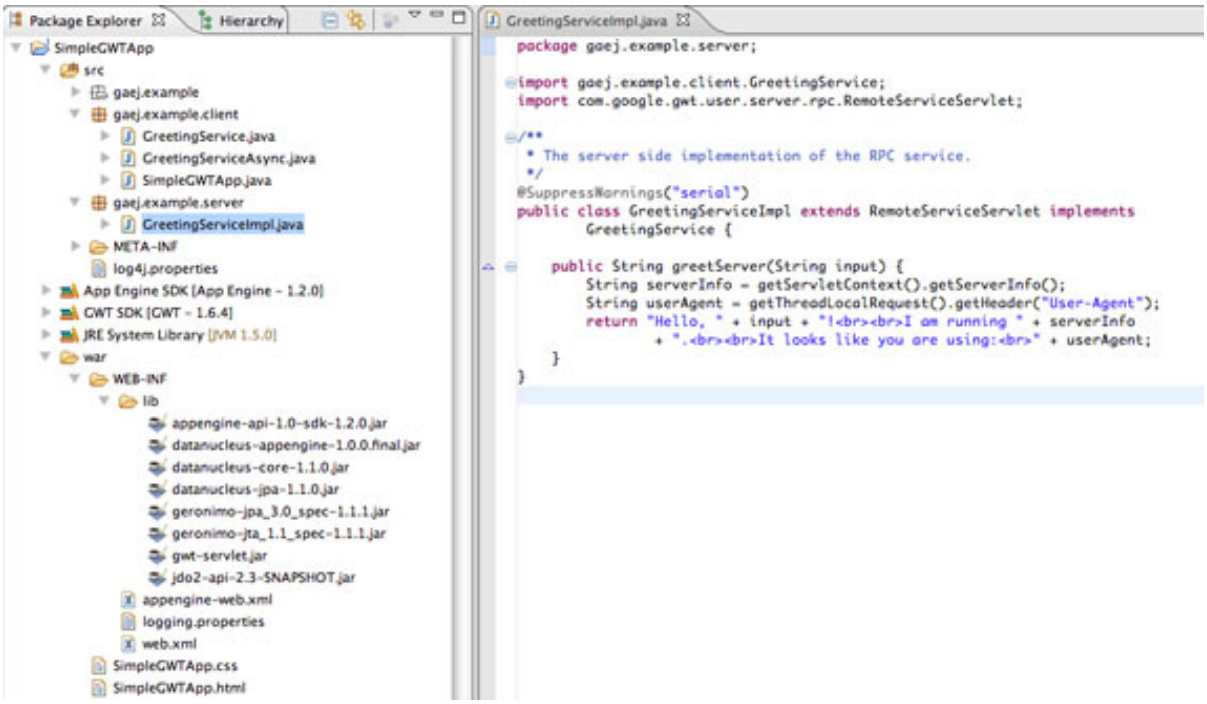
You've got an idea of how a simple App Engine for Java servlet application works, so let's explore the App Engine for Java Eclipse tooling for GWT applications next. Start by clicking the blue ball in your Eclipse IDE toolbar to activate the Google project-creation wizard. This time, select support for GWT, as shown in Figure 5:

Figure 5. Creating a simple GWT application with the App Engine for Java project-creation wizard



As you can see in Figure 6, App Engine for Java provides a lot more code artifacts for a GWT application than for a simple servlet-based one. The example application is a GUI done in GWT that talks to a greeting service application.

Figure 6. Code artifacts provided for a GWT application



There's an extra JAR for the GWT app that wasn't necessary for the servlet-based one, namely gwt-servlet.jar.

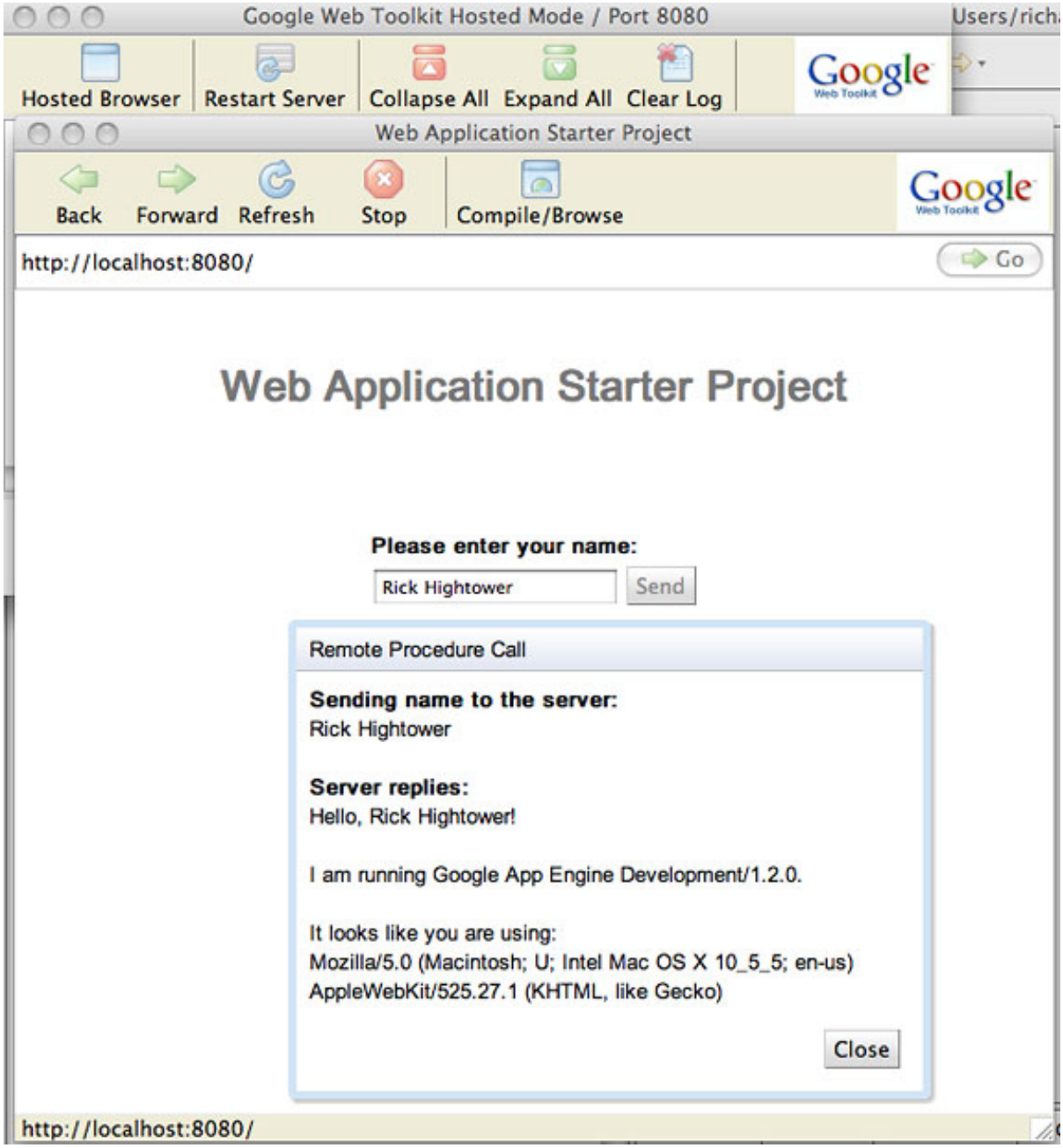
The other artifacts are as follows:

- src/gaej/example: SimpleGWTApp.gwt.xml: GWT module descriptor
- src/gaej.example.server: GreetingServiceImpl.java: Implementation of the greeting service
- src/gaej.example.client: GreetingService.java: Synchronous API for the greeting service
- src/gaej.example.client: GreetingServiceAsync.java: Asynchronous API for the greeting service
- src/gaej.example.client: SimpleGWTApp.java: Main entry point that also builds the starter GUI
- war/WEB-INF: web.xml: Deployment descriptor that configures `GreetingServiceImpl`
- war: SimpleGWTApp.html: HTML page that displays the GWT GUI
- war: SimpleGWTApp.css: Stylesheet for the GWT GUI

Before you drill down to the application's architecture and source code, see what happens when you run it. To run the application, click the red toolbox on your

toolbar, then click the **Compile** button. Now right-click the project and select the **Run As—> Web Application** menu item like you did before. This time, because you're working on a GWT application, a GWT Hosted Mode Console and browser will appear. Go ahead and use the Web application to enter your name and see the response. I received the response shown in Figure 7:

Figure 7. Running the sample GWT application



In the next sections, I'll walk you through the example GWT application. If you want to know more about GWT (or take a GWT tutorial), see Resources.

Inside the GWT app

Based on the provided configuration, Eclipse's GWT tooling creates a starter application featuring an HTML front end (SimpleGWTApp.html, shown in Listing 10) that loads simplegwtapp.js and simplegwtapp.nocache.js. This is JavaScript code that is generated by GWT from your Java code; namely, the code that is in the src directory under the gaej.example.client package (see Listings 6, 7, and 8).

The main entry point for GUI creation is `gaej.example.client.SimpleGWTApp`, shown in Listing 8. This class creates GWT GUI elements and associates them with HTML DOM elements on the SimpleGWTApp.html (see Listing 10). The SimpleGWTApp.html defines two DOM elements named `nameFieldContainer` and `sendButtonContainer` (columns in a table). The `SimpleGWTApp` class uses `RootPanel.get("nameFieldContainer")` to access the panel associated with those DOM elements and replace them with GUI elements. The `SimpleGWTApp` class then defines a textbox and button, which you can use to input someone's name and send them a greeting (see Listing 10).

GWT knows that the `SimpleGWTApp` class is the main entry point for the application because SimpleGWTApp.gwt.xml specifies it as such with the entry-point element.

`SimpleGWTApp` wires up the button, called `sendButton`, so that when it is clicked `SimpleGWTApp` will invoke the `greetServer` method on `GreetingService`. The `GreetingService` interface is defined in src/gaej.example.client.GreetingService.java (Listing 6).

Because Ajax is inherently asynchronous, GWT defines an asynchronous interface for accessing remote services. The SimpleGWTApp uses the asynchronous interface defined in src/gaej.example.client.GreetingServiceAsync.java (Listing 7). The `GreetingServiceImpl` (src/gaej.example.server.GreetingServiceImpl.java) implements the `greetServer` method defined in the `GreetingService` (Listing 5). The `GreetingServiceImpl.greetServer` method returns a greeting message `String` that the SimpleGWTApp uses to display the greeting message in the dialog box it creates.

The GWT module descriptor declares the main entry point for the GUI application, namely `gaej.example.client.SimpleGWTApp`, as shown in Listing 4:

Listing 4. GWT module descriptor (src/gaej/example/SimpleGWTApp.gwt.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.6.4//EN"

"http://google-web-toolkit.googlecode.com/svn/tags/1.6.4/
  distro-source/core/src/gwt-module.dtd">


<module rename-to='simplegwtapp'>
```

```
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.  You can change       -->
  <!-- the theme of your GWT application by uncommenting          -->
  <!-- any one of the following lines.                            -->
  <inherits name='com.google.gwt.user.theme.standard.Standard'/>
  <!-- <inherits name='com.google.gwt.user.theme.chrome.Chrome'/> -->
  <!-- <inherits name='com.google.gwt.user.theme.dark.Dark'/>     -->

  <!-- Other module inherits                                      -->

  <!-- Specify the app entry point class.                         -->
  <entry-point class='gaej.example.client.SimpleGWTApp'/>
</module>
```

`GreetingServiceImpl` is the actual implementation of the greeting-service application, shown in Listing 5. It runs on the server side and the client code calls it via a remote procedure call.

Listing 5. Implementation of the greeting-service app (src/gaej.example.server.GreetingServiceImpl.java)

```
package gaej.example.server;

import gaej.example.client.GreetingService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
 * The server side implementation of the RPC service.
 */
@SuppressWarnings("serial")
public class GreetingServiceImpl extends RemoteServiceServlet implements
        GreetingService {

    public String greetServer(String input) {
        String serverInfo = getServletContext().getServerInfo();
        String userAgent = getThreadLocalRequest().getHeader("User-Agent");
        return "Hello, " + input + "!<br><br>I am running " + serverInfo
                + ".<br><br>It looks like you are using:<br>" + userAgent;
    }
}
```

`GreetingService`, shown in Listing 6, is the interface for the remote procedure call used by the client code:

Listing 6. Synchronous API (src/gaej.example.client.GreetingService.java)

```
package gaej.example.client;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

/**
 * The client side stub for the RPC service.
 */
@RemoteServiceRelativePath("greet")
public interface GreetingService extends RemoteService {
```

```
    String greetServer(String name);
}
```

`GreetingServiceAsync` is the actual interface that the client code will use, as shown in Listing 7. Each method provides a callback object so that you are notified asynchronously when the remote procedure call is complete. Under the hood, GWT uses Ajax. When using Ajax on the client, it is best if you don't block the client, and thus the asynchronous calls. Blocking would defeat the purpose of using Ajax.

Listing 7. Asynchronous API (src/gaej.example.client.GreetingServiceAsync.java)

```
package gaej.example.client;

import com.google.gwt.user.client.rpc.AsyncCallback;

/**
 * The async counterpart of <code>GreetingService</code>.
 */
public interface GreetingServiceAsync {
    void greetServer(String input, AsyncCallback<String> callback);
}
```

The `SimpleGWTApp` is where most of the action happens. It registers for GUI events, then sends Ajax requests to the `GreetingService`.

Listing 8. This main entry point to the application also builds the starter GUI (src/gaej.example.client.SimpleGWTApp.java)

```
package gaej.example.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DialogBox;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class SimpleGWTApp implements EntryPoint {
    /**
     * The message displayed to the user when the server cannot be reached or
     * returns an error.
     */
    private static final String SERVER_ERROR = "An error occurred while "
```

```
            + "attempting to contact the server. Please check your network "
            + "connection and try again.";

    /**
     * Create a remote service proxy to talk to the server-side Greeting service.
     */
    private final GreetingServiceAsync greetingService = GWT
            .create(GreetingService.class);

    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {
        final Button sendButton = new Button("Send");
        final TextBox nameField = new TextBox();
        nameField.setText("GWT User");

        // You can add style names to widgets
        sendButton.addStyleName("sendButton");

        // Add the nameField and sendButton to the RootPanel
        // Use RootPanel.get() to get the entire body element
        RootPanel.get("nameFieldContainer").add(nameField);
        RootPanel.get("sendButtonContainer").add(sendButton);

        // Focus the cursor on the name field when the app loads
        nameField.setFocus(true);
        nameField.selectAll();

        // Create the popup dialog box
        final DialogBox dialogBox = new DialogBox();
        dialogBox.setText("Remote Procedure Call");
        dialogBox.setAnimationEnabled(true);
        final Button closeButton = new Button("Close");
        // You can set the id of a widget by accessing its Element
        closeButton.getElement().setId("closeButton");
        final Label textToServerLabel = new Label();
        final HTML serverResponseLabel = new HTML();
        VerticalPanel dialogVPanel = new VerticalPanel();
        dialogVPanel.addStyleName("dialogVPanel");
        dialogVPanel.add(new HTML("<b>Sending name to the server:</b>"));
        dialogVPanel.add(textToServerLabel);
        dialogVPanel.add(new HTML("<br><b>Server replies:</b>"));
        dialogVPanel.add(serverResponseLabel);
        dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
        dialogVPanel.add(closeButton);
        dialogBox.setWidget(dialogVPanel);

        // Add a handler to close the DialogBox
        closeButton.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                dialogBox.hide();
                sendButton.setEnabled(true);
                sendButton.setFocus(true);
            }
        });

        // Create a handler for the sendButton and nameField
        class MyHandler implements ClickHandler, KeyUpHandler {
            /**
             * Fired when the user clicks on the sendButton.
             */
            public void onClick(ClickEvent event) {
                sendNameToServer();
            }

            /**
             * Fired when the user types in the nameField.
             */
```

```
            public void onKeyUp(KeyUpEvent event) {
                if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
                    sendNameToServer();
                }
            }

            /**
             * Send the name from the nameField to the server and wait for a response.
             */
            private void sendNameToServer() {
                sendButton.setEnabled(false);
                String textToServer = nameField.getText();
                textToServerLabel.setText(textToServer);
                serverResponseLabel.setText("");
                greetingService.greetServer(textToServer,
                        new AsyncCallback<String>() {
                            public void onFailure(Throwable caught) {
                                // Show the RPC error message to the user
                                dialogBox
                                        .setText("Remote Procedure Call - Failure");
                                serverResponseLabel
                                        .addStyleName("serverResponseLabelError");
                                serverResponseLabel.setHTML(SERVER_ERROR);
                                dialogBox.center();
                                closeButton.setFocus(true);
                            }

                            public void onSuccess(String result) {
                                dialogBox.setText("Remote Procedure Call");
                                serverResponseLabel
                                        .removeStyleName("serverResponseLabelError");
                                serverResponseLabel.setHTML(result);
                                dialogBox.center();
                                closeButton.setFocus(true);
                            }
                        });
            }
        }

        // Add a handler to send the name to the server
        MyHandler handler = new MyHandler();
        sendButton.addClickHandler(handler);
        nameField.addKeyUpHandler(handler);
    }
}
```

The Web deployment descriptor (web.xml, shown in Listing 9) maps `GreetingService` as a servlet-based Web resource. It maps the `GreetingService` servlet under the name `/simplegwtapp/greet` so that the `SimpleGWTApp` can load it and make calls to it. The Web deployment descriptor also denotes that SimpleGWTApp.html is the welcome page for the application, so that it always loads.

Listing 9. Deployment descriptor that configures GreetingServiceImpl (war/WEB-INF/web.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>

  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>SimpleGWTApp.html</welcome-file>
  </welcome-file-list>

  <!-- Servlets -->
  <servlet>
    <servlet-name>greetServlet</servlet-name>
    <servlet-class>gaej.example.server.GreetingServiceImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>greetServlet</servlet-name>
    <url-pattern>/simplegwtapp/greet</url-pattern>
  </servlet-mapping>

</web-app>
```

The HTML front end is SimpleGWTApp.html, shown in Listing 10. This is the page that loads simplegwtapp.js and simplegwtapp.nocache.js, which is JavaScript code generated by GWT from your Java code. As previously mentioned, this code lives in the src directory under the gaej.example.client package (from Listings 6, 7, and 8).

Listing 10. The HTML page that displays the GWT GUI (war/SimpleGWTApp.html)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- The HTML 4.01 Transitional DOCTYPE declaration-->
<!-- above set at the top of the file will set     -->
<!-- the browser's rendering engine into           -->
<!-- "Quirks Mode". Replacing this declaration     -->
<!-- with a "Standards Mode" doctype is supported, -->
<!-- but may lead to some differences in layout.   -->

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">

    <!--                                                               -->
    <!-- Consider inlining CSS to reduce the number of requested files -->
    <!--                                                               -->
    <link type="text/css" rel="stylesheet" href="SimpleGWTApp.css">

    <!--                                           -->
    <!-- Any title is fine                         -->
    <!--                                           -->
    <title>Web Application Starter Project</title>

    <!--                                           -->
    <!-- This script loads your compiled module.   -->
    <!-- If you add any GWT meta tags, they must   -->
    <!-- be added before this line.                -->
    <!--                                           -->
    <script type="text/javascript" language="javascript"
  src="simplegwtapp/simplegwtapp.nocache.js"></script>
  </head>

  <!--                                        -->
  <!-- The body can have arbitrary html, or   -->
  <!-- you can leave the body empty if you want  -->
```

```
  <!-- to create a completely dynamic UI.        -->
  <!--                                           -->
  <body>

    <!-- OPTIONAL: include this if you want history support -->
    <iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
  style="position:absolute;width:0;height:0;border:0"></iframe>

    <h1>Web Application Starter Project</h1>

    <table align="center">
      <tr>
        <td colspan="2" style="font-weight:bold;">Please enter your name:</td>
      </tr>
      <tr>
        <td id="nameFieldContainer"></td>
        <td id="sendButtonContainer"></td>
      </tr>
    </table>
  </body>
</html>
```

With GWT, you control the look and feel of your app through CSS, as demonstrated in Listing 11:

Listing 11. Stylesheet for the GWT GUI (war/SimpleGWTApp.css)

```
/** Add css rules here for your application. */


/** Example rules used by the template application (remove for your app) */
h1 {
  font-size: 2em;
  font-weight: bold;
  color: #777777;
  margin: 40px 0px 70px;
  text-align: center;
}

.sendButton {
  display: block;
  font-size: 16pt;
}

/** Most GWT widgets already have a style name defined */
.gwt-DialogBox {
  width: 400px;
}

.dialogVPanel {
  margin: 5px;
}

.serverResponseLabelError {
  color: red;
}

/** Set ids using widget.getElement().setId("idOfElement") */
#closeButton {
  margin: 15px 6px 6px;
}
```

Deploying to Google App Engine

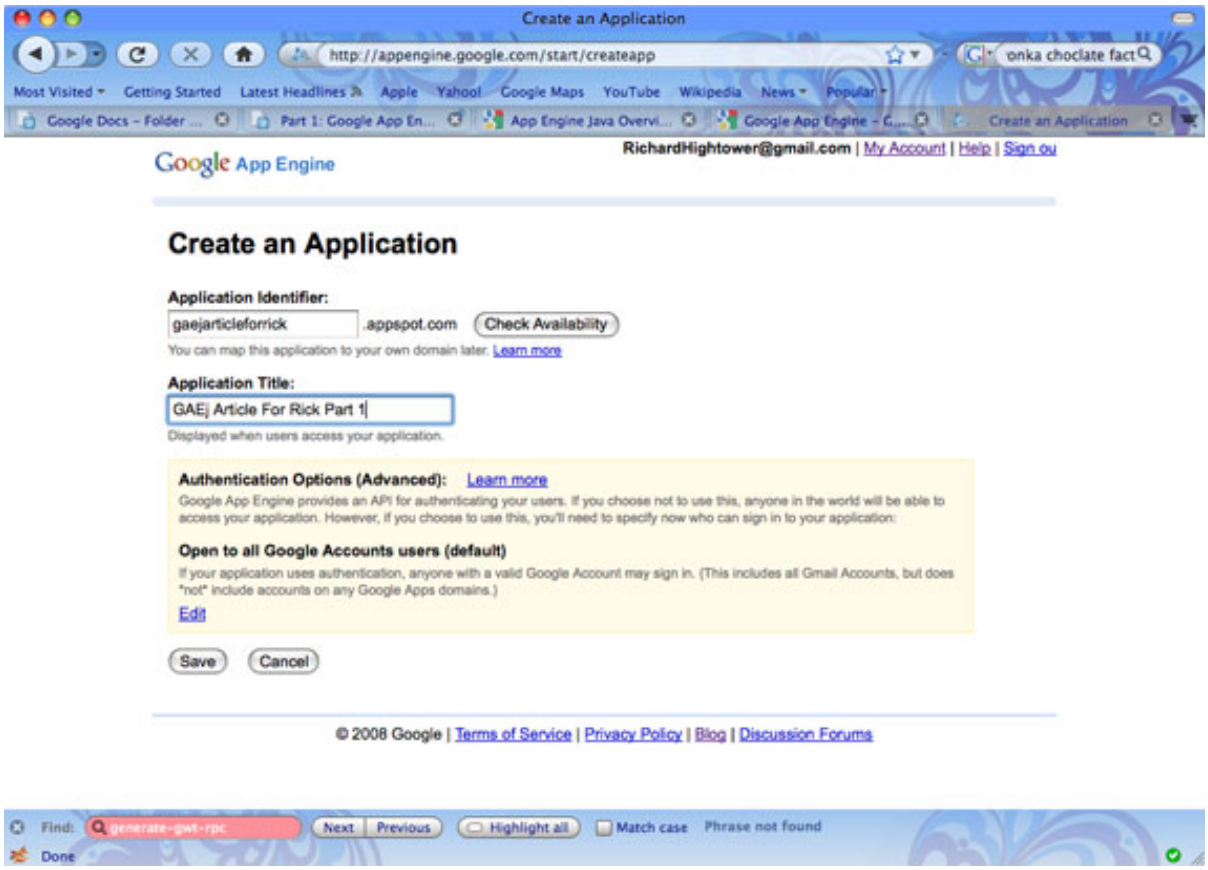
Once you have created the world's next killer application (because we really need a user-friendly greeting application), you'll want to deploy it. The whole point of using Google App Engine is that you can deploy your application on Google's solid infrastructure, making it easier to scale. Google App Engine is designed to provide a platform for building scalable applications "that grow from one to millions of users without infrastructure headaches" (as stated on the App Engine home page). In order to use this infrastructure, you need a Google App Engine for Java account.

Like many things in life, the first time is free. The free version of App Engine for Java gives a deployed application enough CPU, bandwidth, and storage to serve about 5 million pageviews. Beyond that, it's pay as you go. (Also keep in mind that what's available as of this writing is a preview release of the App Engine for Java platform.)

Once you get the account, you should see an empty list of applications at the App Engine for Java site. Click the **Create New Application** button and a form like the one in Figure 8 should appear. Enter a unique application name and a description, after which you will see a confirmation message with your application's identifier.

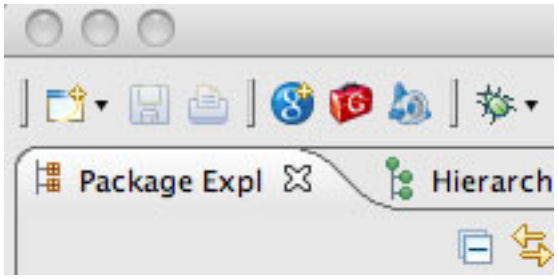
The identifier belongs in your application's app.yaml file as well. Note that the identifier cannot be changed. If you use Google authentication for your application, "GAEj Article For Rick Part 1" will be displayed in the Sign In pages when you access your application. You'll use `gaejarticleforrick` to deploy the application to Google App Engine with the App Engine for Java Eclipse plugin.

Figure 8. Creating a new App Engine for Java application



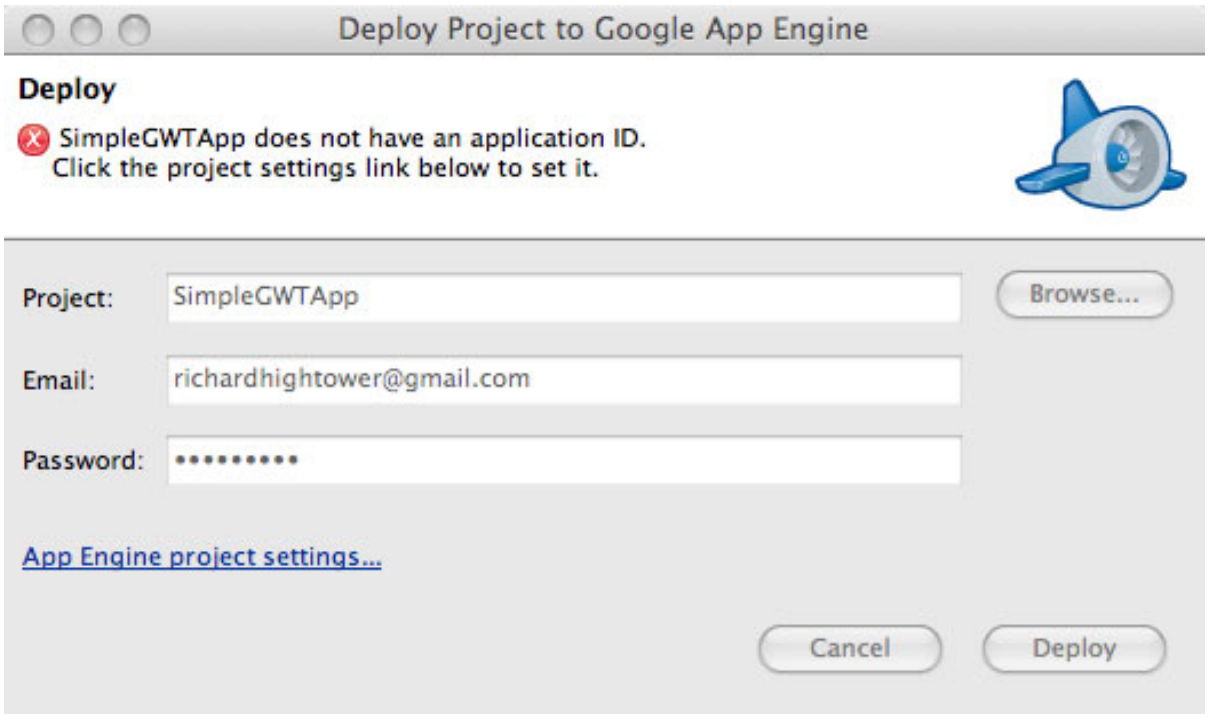
After you set up the application ID, you can deploy your application from Eclipse. First, hit the toolbar button that looks like the Google App Engine logo (a jet engine with wings and tail), shown in Figure 9:

Figure 9. App Engine for Java Eclipse plugin



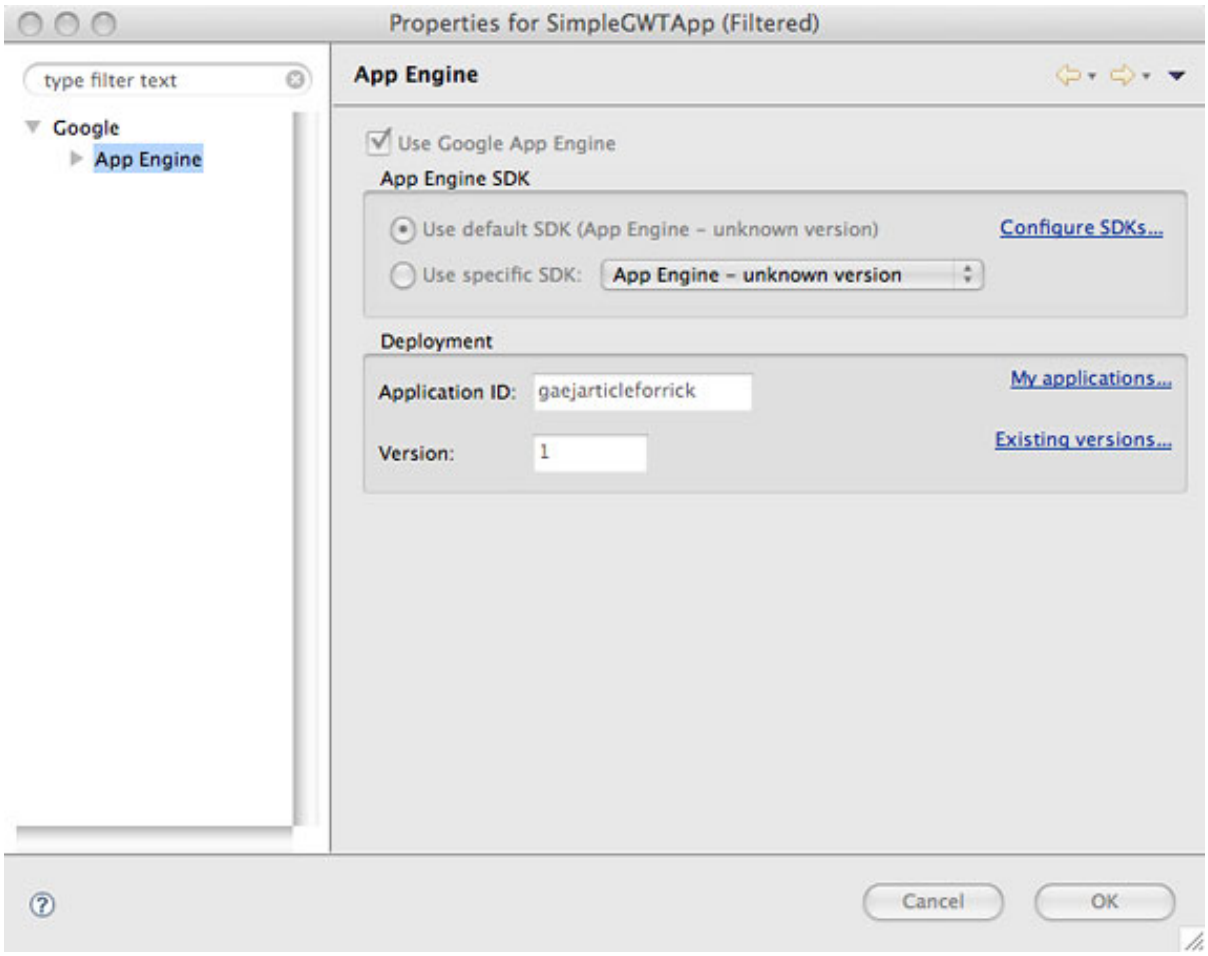
You may need to make sure your App Engine for Java project is selected before clicking **Deploy** in the dialog shown in Figure 10. You will be prompted for your Google credentials, which are your e-mail address and username.

Figure 10. Deploying the project



The dialog in Figure 10 has a link to "App Engine Project setting." Click this link (also accessible from the project settings file) and enter in the application ID (in this case `gaejarticleforrick`), as show in Figure 11. After you fill in the application ID, click **OK**, then click **Deploy**.

Figure 11. Project setting for Google App Engine



After you have deployed your application, it will be available at `http://<application id>.appspot.com/`. You can also see the application live at http://gaejarticleforrick.appspot.com/.

Conclusion

This concludes Part 1 of my introduction to Google App Engine for Java. So far, you've gained an overview of what App Engine for Java is all about and taken first steps with using the App Engine for Java Google Plugin for Eclipse. You created two small starter applications (one servlet-based and the other based on GWT) and then deployed the GWT application to the Google App Engine platform.

The examples so far have demonstrated the tooling and functionality that make it easier to create and deploy Java-based applications that scale — potentially even to the size of YouTube or Facebook. In Part 2, you'll continue to explore the opportunities available to Java developers working on App Engine for Java. Moving away from the example apps demonstrated in this article, you'll build a custom contact management application. This application will also be the centerpiece of the

exercises in Part 3, which delve into App Engine for Java's datastore and its GUI front end.

Resources

Learn

- "Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 1: The fancy front end" (Noel Rappin, developerWorks, October 2006): Launches a four-part tutorial introduction to Ajax programming with Google Web Toolkit.
- "Connecting to the cloud, Part 1: Leverage the cloud in applications" (Mark O'Neill, developerWorks, April 2009): A three-part overview of cloud computing platforms from the major vendors: Amazon, Google, Microsoft®, and SalesForce.com.
- DataNucleus project: Provides products for managing application data in a Java environment.
- Google App Engine homepage: Learn more about App Engine.
- Google App Engine Java docs: More about Java development with App Engine for Java.
- Will it play in App Engine for Java?: Find out what standard Java APIs and frameworks are compatible with App Engine for Java.
- What's BigTable?: Read the Google research publication to find out.
- "GWT: The most important announcement at JavaOne?" (Rick Hightower, Java Developers Journal, June 2006): More about why GWT is also a step forward for Java application development.

Get products and technologies

- App Engine for Java Eclipse plugins: Download the plugins to get started.
- App Engine account: You'll need one to deploy your killer app using the Google App Engine infrastructure.

Discuss

- App Engine for Java discussion group: Subscribe to this group to give and receive help and advice as you learn about App Engine for Java.
- Get involved in the My developerWorks community.

About the author

Rick Hightower
Rick Hightower serves as chief technology officer for Mammatus Inc., a training

company that specializes in cloud computing, GWT, Java EE, Spring, and Hibernate development. He is coauthor of the popular book *Java Tools for Extreme Programming* and author of the first edition of *Struts Live* — the number-one download on TheServerSide.com for many years. He is has also written articles and tutorials for IBM developerWorks and is on the editorial board for Java Developer's Journal, as well as a frequent contributor to the Java and Groovy topics on DZone.