Extracted from:

Code in the Cloud

Programming Google AppEngine

This PDF file contains pages extracted from Code in the Cloud, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Code in the Cloud

Programming Google AppEngine

Mark C. Chu-Carroll

Edited by Colleen Toporek

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Copyright © 2010 Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-63-8

ISBN-13: 978-1-934356-63-0

Printed on acid-free paper.

B1.0 printing, March 17, 2010

Version: 2010-3-15

your program. This is particularly valuable in an environment like the cloud, where it's harder to debug your program. You can't just fire up a debugger and probe it. You can't add print statements to find where things went wrong. Anything that helps you catch problems ahead of time can be a huge time-saver.

Style. As you'll see later in this chapter, developing a cloud application in Java has a very different style and structure from Python. For some developers, the style of Java development in AppEngine can be much more comfortable than Python.

Tools. Google released a set of plugins for the free Eclipse IDE for building Java/GWT AppEngine services and applications. Eclipse is an absolutely *amazing* tool, and the AppEngine plugins make everything easier. (You can use Eclipse with Python, but there's no specific AppEngine support, so it ends up being pretty painful.)

In this chapter, we'll take a look at developing cloud applications using GWT. We'll do that by taking our chat application, and porting it to Java/GWT. We'll go through a compressed version of our journey so far, looking at how to do what we've already done, this time in Java.

9.1 Introducing GWT

There's one reason for using Java that completely outweighs all of the others: GWT. GWT is amazing. It lets you write your entire cloud application in Java. The server side is compiled in the usual way for Java: compiled into Java bytecodes that are executed on the JVM. On the server side, it's a nice framework, but it's not particularly *special*. But then there's the client: GWT lets you write your client as a Java program. You write the client in Java almost like a traditional GUI application: you build a UI from a collection of widgets using layout managers, attach event handlers, and so on—absolutely typical GUI code. But GWT translates that GUI code into HTML and JavaScript: instead of compiling Java to Java bytecodes, it compiles Java *to JavaScript source code*, which then executes on the client. And for all of the AJAX stuff in which the client and server needs to communicate, GWT can generate remote procedure calls. It's not a totally automatic process, but it's vastly easier and more robust than writing JavasScript AJAX code manually. (To be honest, my first reaction when I heard about this was, "They're out of their minds; that's ridiculous!". Which goes to show you why I'm not rich and famous.)

CLICK HERE to purchase this book now.

Because of the way it's set up, building an application in GWT is different from what we did in Python with webapp. Our first example is going to have a beautiful UI; we don't need to wait to get to how to set up templates and floats with CSS—we'll just dive right in, and let GWT do what it does best.

Programming in GWT is, in many ways, much more like programming an application with a traditional desktop GUI framework. You define your UI almost the same way you would for a traditional desktop app, and GWT takes care of generating most of the HTML, CSS, and JavaScript that's necessary for making that app work. Most of Google's recent applications (including things like Wave) are implemented using GWT.

To start looking at GWT, download the AppEngine SDK for Java. I'm not going to walk through it in detail, because it's basically the same process that you used to download the Python SDK in Chapter 2, *Getting Started*, on page 18. In addition to the basic framework, you can also install a set of plugins for Eclipse, which provide an excellent programming environment. I highly recommend downloading Eclipse and the AppEngine plugins. The ability to use Eclipse for AppEngine development is one of the best reasons for working with Java! Eclipse is free, and it's really easy to set up. The downside to GWT is that there's a lot of *metadata*; that is, a lot of extra files that tell GWT what to do with the Java source, things like which parts to compile to JavaScript for the client, which parts to set up as a servlet bundle for the server, and so on. Maintaining all of those files can be painful, but the Eclipse tooling is a huge help. You *can* program in GWT without using Eclipse, but you really shouldn't. From here on, I'm going to assume that you're using Eclipse with the GWT plugins.

GWT constitutes a very different approach to building a cloud application. In Python and webapp, everything was focused on the server. Of course, we built client UIs, but we did it by focusing on what *the server needed to do* to generate the UI on the client. The process centered on building request handlers, and the CSS and templates that the request handlers needed. GWT is almost exactly opposite: in GWT, you focus on the client. You build a client UI using a framework that looks like a traditional client application. When your client needs something from the server, you make a *remote procedure call (RPC)* to invoke it; GWT takes care of most of the work of turning that RPC into an AJAX call.

With that in mind, let's start building a GWT application.

CLICK HERE to purchase this book now.

9.2 Getting Started with Java and GWT

To begin, we'll look at something like a basic "Hello World" program. The GWT tools for Eclipse automatically build a project skeleton, which is a basic GWT hello-world; so instead of writing our own, we'll just let Eclipse do it, and walk through the pieces, seeing how it's all put together. In Eclipse, select "New" from the "File" menu. In the dialog that comes up, pick "New Web Application Project". Then fill in the resulting dialog box with a project name, and the name of the Java package you want to use for your Java code. I selected "HelloChat" as the project name, and "com.pragprog.aebook.hellochat" for the Java package name.

The starter application sets up a page that prompts users for their name; when users enter their names, it pops up a dialog box saying hello to them.

The Structure of a GWT Application

A GWT application consists of a set of *modules*. A module is a GWT package consisting of Java code, JavaScript, HTML files, images, data definitions, and whatever else you need in a web application. The directory structure that you get when you create a GWT/AppEngine project in Eclipse is based on the structure of the GWT module that it implements.

To begin with, let's look at that directory structure. You can see the structure in the Eclipse package browser in Figure 9.1, on the following page. Inside the AppEngine project, there are a collection of GWT libraries, plus two main components: a source directory named src, and a target directory named war. "war" stands for "web archive": the deployable application that you upload to app-engine is a war file.

The source directory itself is also divided into three parts: a *module declaration*, a package for the client-side Java code, and a package for the server-side Java code.

The server package, com.pragprog,aebook.hellochat.server, is deceptively simple, consisting of one, almost trivial source file, because GWT is going to automatically generate the server-side plumbing.

The client side has a three files. One of them, HelloChat.java is the main body of our application. The other two, GreetingService.java and GreetingServiceImpl.java are part of the setup for a GWT remote procedure call. These files contain the declarations that GWT needs in order to allow

CLICK HERE to purchase this book now.

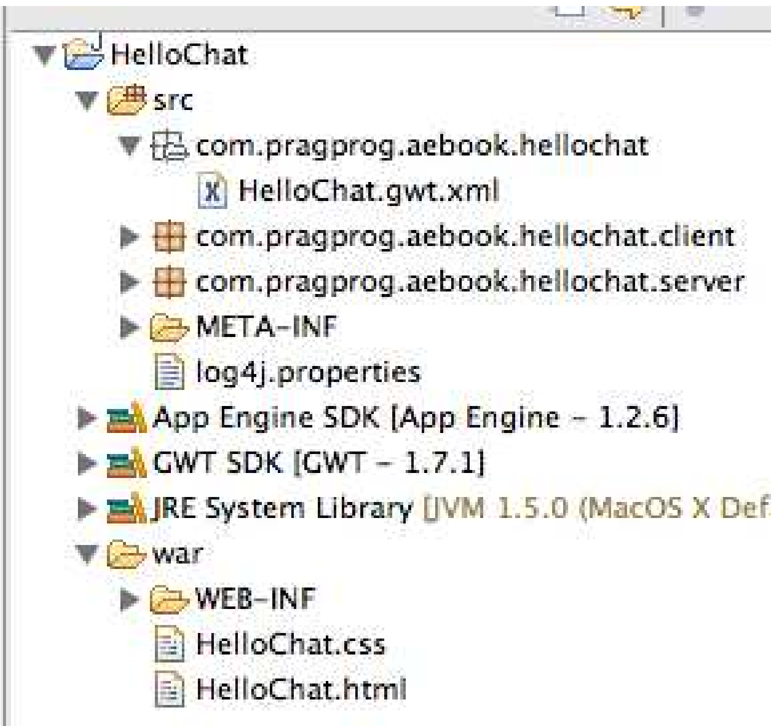

Figure 9.1: The GWT project directory structure in Eclipse

us to do AJAX client/server applications without explicitly setting up XMLHttpRequests. We'll look at how those files work in Section 9.3, *RPC in GWT*, on page 132.

The way that these pieces fit together is determined by the GWT module declaration.

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/HelloChat.gwt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.7.1//EN"
    "http://google-web-toolkit.googlecode.com/.../gwt-module.dtd">
❶ <module rename-to='hellochat'>

  <inherits
❷        name='com.google.gwt.user.User'/>

  <inherits
❸        name='com.google.gwt.user.theme.standard.Standard'/>
```

```
  <entry-point
❹   class='com.pragprog.aebook.hellochat.client.HelloChat'/>
</module>
```

❶ The fundamental unit of code in GWT is a *module*. A module consists of a collection of things: Java code; resources like CSS, HTML, or image files; and GWT customizations, like Java to JavaScript compiler extensions. This line declares the module that will contain our application. The "rename" element is part of GWT's URL handling: GWT will tell the server to set this module up at a URL path ending with "hellochat".

❷ Modules in GWT can *inherit* things from other modules. It works pretty much like object-oriented inheritance. Our application is a *sub-module* of com.google.gwt.user.User, which is the standard module for an application with a user interface. Most of the basic functionality of GWT—the UI widgets, the remote procedure call plumbing, and the basic server-side servlet infrastructure—are inherited through this declaration.

❸ Part of the reason GWT defines modules in addition to using class inheritance in the Java code is because there are a lot of resources in a GWT module besides code. A module can include things like CSS. The inherit statement pulls in the CSS files that define the look of the UI widgets in our application. We can change the look of our application by inheriting from a different style module.

❹ The Java code for a GWT application starts with an *entry point*. An entry point is, pretty much, the GWT GUI equivalent of a main function. In the module file, you declare entry points for code you want executed in your GWT application. In this case, the entry point is the class HelloChat.

Setting Up the UI in GWT

Within a GWT module, the user interface frame is defined by an HTML file. The HTML file isn't considered source code, so it doesn't get put into the src directory. It's a static *resource*: a file that contains information that will be used by the code. So the HTML file ends up in the war directory. Let's take a look at its contents:

Download workspace/HelloChat/war/HelloChat.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

CLICK HERE to purchase this book now.

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

    <link type="text/css" rel="stylesheet"
          href="HelloChat.css"/>

    <title>Web Application Starter Project</title>

    <script type="text/javascript" language="javascript"
❷           src="hellochat/hellochat.nocache.js"></script>
  </head>

  <body>
    <h1>Hello World</h1>

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

❶ The HTML frame file is a standard HTML file. It starts off with the usual HTML stuff: the doctype declaration, the head block with the usual meta-tags.

❷ This is the most important line of the entire file! What makes the HTML file into a GWT application frame is this include line. It pulls in the JavaScript file that's going to be generated by GWT, containing all of our application code.

❸ As I'll explain in more detail later, you can do layout in the UI using either static structures defined in the HTML file, or dynamic structures defined in Java code. For our application, that HTML frame defines a static structure for the main UI page. The easiest way to do that is using HTML tables. (We could also do it using CSS floats, as we saw in the Python code, but if we want to do dynamic layout, it would be much better to let GWT take care of it.) So we set up a two-column table: one column for the text entry box, and one for the "send" button.

Click Here to purchase this book now.

❹ The HTML static structure can include static content as well as static structure. As usual, if we can separate things like static content from program logic, we should. So we use the static frame here to insert a title line, and use the HTML table layout controls to make it spans both columns of the layout.

❺ Now we get to something interesting. What we're doing here is creating an *empty* box in the UI. The <*td*> tag creates a box in the HTML layout, but it's empty—there's nothing inside of the tag. In our Java code, we'll insert something, referencing it using its id= tag. We create two boxes this way: one for the text box, and one for the button.

Now we can get to some code. As we saw above in the module declaration, the application has one entry point. The full entry point method is pretty long; it incorporates both the creation of the UI elements, setting up event handlers, and setting up remote procedure calls for the client/server communication. Let's look at it in pieces. We'll start with the part that builds the main UI; that is, the main page that prompts the users for their names.

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/HelloChat.java

```
❶ public void onModuleLoad() {
❷         final Button sendButton = new Button("Send");
          final TextBox nameField = new TextBox();
          nameField.setText("GWT User");

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

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

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

❶ An entry point class is a container for the GWT equivalent of a "main" function. Conceptually, it really is like the main program in a non-GUI tool. But in Java, everything needs to be enclosed in a class, so we must create a skeleton class around the actual main. In a typical GWT application, this is the *only* method that's defined on the entry point class—it's just an overcomplicated wrapper for a single method. The real main function is the "onModuleLoad"

CLICK HERE to purchase this book now.

method of the entry point. As the name suggests, this is what gets executed when the GWT module is loaded by the client. Inside this method, we create the UI widgets, lay them out, and set up the event handlers.

❷ The first thing we do inside of onModuleLoad is create the UI widgets. For basic cases, it looks pretty much like the way we'd do it if we were building a non-browser UI. We create a button, and a text box where the users will enter their names.

❸ The first place that things start to look different from a traditional non-browser UI is in the management of the style attributes of the widgets. In a typical GUI toolkit, there are a set of methods to call for various style attributes. For example, in the Mac OS Cocoa widgets, we could modify the gradient of a button using a call like [button setGradientType: NSGradientConcaveWeak]. In GWT, that's all done using CSS: we'd set a CSS attribute to create a gradient image for the button background; we'd add the line background: url("images/gradient.png") to the CSS style block for .gwt-Button. The only call for managing style is one that sets up a connection to a CSS style. The style name is translated by GWT into a CSS class= attribute. It might seem a bit strange at first, but it's really nice in practice: it helps maintain that separation of concerns—you really shouldn't clutter your code with visual style stuff, and you should have all of the style stuff in one place. The way GWT uses CSS gives you a really convenient way of doing that.

❹ Now we get to layout. GWT provides you with a GUI context that's basically the contents of the browser page, called the RootPanel. To access the root panel directly, call RootPanel.get(). We can also do part of our layout using HTML, as in this example. If the application's main HTML page contains elements that are named with an id= attribute, we can access those elements using get(name). In this case, the root page for our application did provide elements for pieces of our application. This is pretty typical of GWT style: we've got a choice between doing things like layout statically (by doing it in HTML), and doing them dynamically (by writing layout code in Java). In general, when the layout is pretty much fixed (like in this case), it's easier to write an HTML table and just fill it in from Java. To create something on the fly, like the dialog box we'll see in a few minutes, use a GWT layout manager. In the static

CLICK HERE to purchase this book now.

layouts, we can get a layout box on the page by calling get, and then inserting a GUI widget into it, using add(widget).

5. Finally, when the UI loads, we'd like it to work so that if the user starts typing, it will show up in the text box. We do that by setting the *focus*: the focus is the widget on the screen that receives UI events like keystrokes. Users can set the focus by clicking the mouse inside of a widget, but it's annoying to be forced to do that when there's only one place where it makes sense for the focus to be. So we set it to focus on the text entry box. We also have it automatically select the place-holder text that we put into the box, so if the users start typing, their text will replace the placeholder.

That's it for the basic building of the GUI.

That leaves us with two other important pieces. Our application is going to get a name from a user, and send it to the server. The server puts that name into a hello message, and sends it back to the client to display in a pop-up dialog box. What we still need to do is put together the client/server communication, and the dialog box. We'll look at the client/server communication in the next section. First, we'll look at the dialog, which is more GWT UI work, but instead of using a static layout from an HTML file, the dialog is fully dynamic.

Download workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/HelloChat.java

```
    // Create the popup dialog box
1   final DialogBox dialogBox = new DialogBox();
    dialogBox.setText("Remote Procedure Call");
    dialogBox.setAnimationEnabled(true);
2   final Button closeButton = new Button("Close");
    // We 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();
3   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);
4   dialogBox.setWidget(dialogVPanel);

5   closeButton.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                    dialogBox.hide();
```

```
                sendButton.setEnabled(true);
                sendButton.setFocus(true);
        }
});
```

❶ First, we need to create the dialog box. This is a popup, so it's not contained in the browser frame. That means that we can't just grab the RootPanel; we need to create a free-standing widget. In GWT, that's easy: DialogBox is a free-standing window frame that can embed any GWT widget—we just create its contents, and insert them. Since it's a window, it has a title bar, and we can set its contents using its setText method.

❷ We want the users to be able to get rid of the dialog box whenever they want, so we create a close button, which we'll add to the dialog box frame later. As usual, we can set the attributes of the widget with CSS. In this case, we do it by diving down directly to the HTML. Given any widget, we can get the XML element corresponding to that widget by calling getElement(). Then we set its ID, to allow a CSS style to reference it, using the setId() method of the XML element.

After the close button, create another couple of widgets. There's a Label, which is a piece of non-editable text embedded in a widget. Then there's something interesting: an HTML widget, which is a wrapper for a chunk of literal HTML text. Whatever is inside of the HTML widget is rendered directly into the HTML page for the UI. That's useful for embedding things like styled text, where it's often easier to just use HTML markup around a piece of text than it would be to do the programmatic manipulation to produce the same effect.

❸ Now, we're going to lay out a series of elements. Since we don't have a static HTML frame, we need to specify how to lay them out using GWT. The layout is pretty simple: it's just a bunch of stuff stacked vertically. GWT has a widget for doing that: the VerticalPanel. We just add the widgets of the UI to the panel in order. Notice the HTML markup here: there's some text we want to show in boldface. Instead of creating a label widget and setting its style attributes to make it bold, we can just wrap the text in <b> tags.

❹ We've got the UI elements laid out in a VerticalPanel. All we need to do is tell the dialog box that the panel is what it should show: we do that by setting the dialog box's widget. Now the visual parts

of the box are all done. The box starts off invisible: standalone widgets like this don't actually appear on the users' screen until we explicitly tell them to. As we'll see later, we can do that with a dialog box by telling it *where* it should appear. Most of the time, that's in the center of the browser window—so the dialog will be made visible by calling its center() method.

❺ With the basic UI set up, we can finally look at how to handle events in GWT! It's pretty much the same as in Java's Swing library. Create a handler object, and attach it to the appropriate widget using an addXXXHandler method. In this case, we're attaching the handler that closes the dialog box when the user clicks its close button, so we attach a ClickHandler object. In its onClick method, we make the dialog box invisible, and enable the entry area of the main page.

9.3 RPC in GWT

Now we get to the complicated part.

As I mentioned before, AJAX code is not written explicitly in GWT. Instead, we write something called a *remote procedure call* (RPC). An RPC is something that looks *almost* like a normal method call, but under the covers, it's translated by the system into a request sent from the client to the server. The return value of the RPC is the response sent from the server back to the client.

Just like any other RPC system, there's a client side and a server side in GWT. We can look at the code for them separately; it's up to the GWT RPC system to string them together.

If you've done any distributed programming, Google-style RPC is probably not what you're used to. Traditionally, RPC tries to appear as much like a traditional function call as possible. In other words, if we want to provide an RPC for a factorial function, the function implementation would look like a traditional function declaration, and an invocation of it would look like a traditional invocation. For example, Java has a native RPC layer, where we define a remote object by an interface, and then we can invoke methods on an object of the interface type.

We could define a factorial service as a Java interface:

```
public interface Fact extends Remote {
    int fact(int n);
```

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Code in the Cloud's Home Page
http://pragprog.com/titles/mcappe
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/mcappe.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)