NEXT

**Google Web Toolkit for Ajax**

By Bruce W. Perry

.....

Publisher: O'Reilly
Pub Date: December 01, 2006
ISBN-10: 0-596-51022-5
ISBN-13: 978-0-596-51022-0
Pages: 56

Table of Contents

## Overview

The Google Web Toolkit (GWT) is a nifty framework that Java programmers can use to create Ajax applications. The GWT allows you to create an Ajax application in your favorite IDE, such as IntelliJ IDEA or Eclipse, using paradigms and mechanisms similar to programming a Java Swing application. After you code the application in Java, the GWT's tools generate the JavaScript code the application needs.

You can also use typical Java project tools such as JUnit and Ant when creating GWT applications. The GWT is a free download, and you can freely distribute the client- and server-side code you create with the framework. This shortcut explains how to get started with the GWT, and then demonstrates how to create a simple Ajax application.

NEXT

**Google Web Toolkit for Ajax**

By Bruce W. Perry

Publisher: O'Reilly
Pub Date: December 01, 2006
ISBN-10: 0-596-51022-5
ISBN-13: 978-0-596-51022-0
Pages: 56

Table of Contents

# Copyright

# Introduction

The term *Ajax* is all the rage in the software world. The acronym originally stood for Asynchronous JavaScript and XML, though now it refers to a broad range of techniques that may not, for example, include XML.

Beyond the technical jargon, Ajax represents an architecture for the software that is designed to be used with any web browser, but that includes an interface with the responsive feel of a desktop application. This means that, for example, a grid component such as a spreadsheet that appears in the web page responds instantly to the user's manipulation of the data, without the time delays or visual disruptions caused by the page "refreshing" or being rebuilt with new HTTP requests. Google's Gmail and Calendar, and Yahoo! Maps, are three examples of typical Ajax applications.

Ajax works by communicating with the application's middle and/or database tiers using a client-side JavaScript object. This object is called XMLHttpRequest (XHR). The web page is composed of and programmed with common, standard technologiesnamely, HTML/XHTML, cascading style sheets (CSS), JavaScript, and the Document Object Model API that lies behind each web page's structure. The data that the page exchanges with its server tiers, such as a product database, can be plain text, XML, or a format called JavaScript Object Notation (JSON).

For example, the user might choose a product name from a selection list widget on a web page. In response to this event, the application uses XHR to send a request for data on this product. The page never changes, because the application sent the request asynchronously behind the scenes. The server component, such as a PHP file or a servlet, receives the HTTP request, then sends back a response containing information about the product, in, say, XML or JSON format. The application processes the request using JavaScript's DOM API, or by converting the JSON value to a client-side object (see later in this shortcut). The user sees the new information on the web page, hopefully without a very long delay.

# Chapter 1. The Google Web Toolkit's Approach to Ajax

A developer will typically create his Ajax application by writing XHTML pages and JavaScript code with his favorite integrated development environment (IDE) or even text editors. A number of different libraries and frameworks exist by which programmers can use pre-designed JavaScript classes to implement otherwise time-consuming dynamic behaviors, such as drag-and-drop or sophisticated visual tree structures. These libraries include the Dojo toolkit, Prototype, the Yahoo! User Interface library, and script.aculo.us. They are designed for developers who are already fairly well advanced in their JavaScript knowledge.

These simple methods are changing as powerful tools proliferate for Ajax developers. The GWT takes a different approach to Ajax than most toolkits.

Using the GWT framework, you can design and program your user interface using only the Java language. You can use the GWT's command-line tools to check the syntax of the Java classes, then automatically generate the JavaScript for the application's client-side. The design of the user interface is very similar to using Java's Swing API.

You can thus view the GWT as a JavaScript-generation tool for Java programmers, as well as a framework for creating redistributable or extensible user-interface widgets. You do not have to know a lick of JavaScript, although you can include raw JavaScript in your code using special programming constructs that the GWT provides.

The GWT's benefits include:

- You can use your favorite tools and existing knowledge to create sophisticated Ajax applications, often without segueing into JavaScript.
- You can design and develop your application in a pure object-oriented fashion, since you are using Java. JavaScript does not have all of Java's built-in OO features.
- Using the GWT removes many of the problems involved with adapting Ajax code to all the different browsers. These problems can represent real hairballs for developers, as two browsers may treat the same piece of code completely differently. You generally do not have to think about these potential incompatibilities while you are designing your application, except for testing your application in the various browsers afterward.
- The JavaScript that the GWT compiler generates is quite obfuscated or very difficult to read, which provides your application with a low level of security or protection of proprietary material. You can "turn off" this obfuscation feature optionally, as this shortcut describes later.
- Google and third parties keep adding libraries to the GWT, or making available ones that are interoperable with the GWT. These activities continue to extend GWT's features and reach. An example is an open-source library called the GWT Widget Library. This JAR file includes classes

that wrap the Scriptaculous Effects objects, allowing you to add these interesting dynamic, visual effects to your GWT application. You can find out more about this library at http://gwt-widget.sourceforge.net.

OK, enough of the high-level GWT discussion. Let's write a program and look at some code!

PREV NEXT

# Chapter 2. Getting Started

First, a few prerequisites. You need an installed Java Development Kit (JDK) of version 1.4 or later. As of this writing, using GWT distribution *gwt-mac-1.2.22*, you cannot use the Java 5 features such as generics or autoboxing with your client-side code, but you can certainly use the Java 5 JDK compiler as you write these classes.

> NOTE
>
> The GWT developers released GWT Release Candidate (RC) 1.3 at the time we published this shortcut. This release is associated with the GWT open-source announcement, however, and did not provide any new functionality. The demo application was compiled and tested with gwt-mac-1.3.1 RC and the official gwt-mac-1.2.22 release.

You have to download and unpack the GWT framework. The download address is http://code.google.com/webtoolkit. You can download the framework for Windows, Linux, or Mac OS X.

The unpacked archive includes:

- Documentation, including a Javadoc format for the GWT API classes;
- The Java Archive (JAR) file (*gwt-user.jar*) containing the user interface classes and other parts of the GWT API (such as modules for handling JSON and XML formats, and making HTTP requests from your client code);
- The JAR file (e.g., *gwt-dev-mac.jar*) containing GWT's proprietary development tools, such as the Java classes that "compile" your code and generate JavaScript;
- The JAR file (*gwt-servlet.jar*) that you can deploy to Tomcat or other servlet containers, and that includes the GWT API for the user-interface widgets and other objects. This is essentially *gwt-user.jar* without the Servlet API classes from the javax package. The Java Servlet Specification, an accepted standard for servlet containers, does not allow packages such as javax.servlet to be loaded from JARs in particular web applications. Therefore, *gwt-servlet.jar* can be placed in *WEB-INF/lib* inside your Java web applications if your server-side code requires the GWT classes to be present on the application's classpath.
- A number of command-line interface tools (see below).

> NOTE

> You do not need to include the *gwt-servlet.jar* file in your Java web application's classpath unless you have server-side classes that use the GWT API, such as the Remote Procedure Call (RPC) related classes and interfaces. This is because the GWT user interface-related Java API is only used in development, before the GWT tools or compilers generate the JavaScript. You then deploy just the generated JavaScript to your web application. The JavaScript is eventually executed inside the browser and no longer has any relation to the GWT Java API (and associated JAR file). By then, it's just JavaScript!

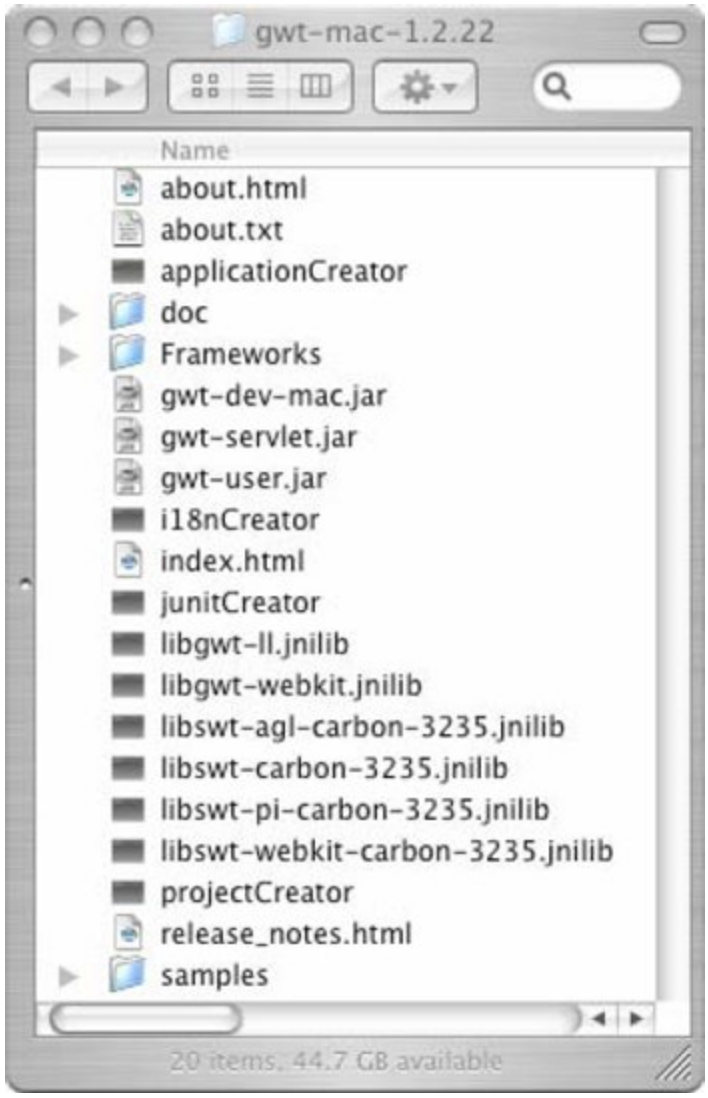
Now we're almost ready to start coding.

PREV NEXT

# Chapter 3. Creating An Application Using the CLI Tools

When you look inside the unpacked directory of the GWT, as in Figure 1, four of the files represent the toolkit's command-line interface (CLI) tools.

Figure 3-1. The GWT's command-line tools inside the unpacked toolkit



The CLI scripts are designed to be run from a command-line window such as Terminal in Mac OS X. They include:

`applicationCreator`

> This generates the skeleton directory structure for your application.

`projectCreator`

> This script generates a project skeleton, as well as Ant build files or Eclipse projects, according to what the command line specifies.

`i18nCreator`

> This creates some of the files required to use internationalized messages with GWT. The shortcut describes this application aspect in another section.

`junitCreator`

> The script can be used to get you started using JUnit with GWT. A later section describes using this command line tool.

We are using `applicationCreator` with our application. I opened up a Terminal or CLI window and typed:

```
applicationCreator -out /users/bruceperry/1ebooks/gwt -overwrite
com.parkerriver.gwt.intro.client.GwtAjax
```

This command uses the provided `applicationCreator` shell script to create a skeleton directory structure. I will use this structure to develop an Ajax application inside a file named *GwtAjax.java*. The first option with the `applicationCreator` command, `-out`, specifies the directory for your application or project (it defaults to the existing directory, the one where `applicationCreator` resides, which is the top-level directory when you unpack GWT).

The `-overwrite` option specifies that the command should overwrite any existing files. The final segment is the fully qualified name of the class where your application logic resides, usually representing the "entry point" class.

An entry point is the first screen the user interacts with in the Ajax application, such as a login window or a dashboard.

NOTE

The `projectCreator` command-line tool will optionally generate an Eclipse project or Ant build file using the `-eclipse` or `-ant` options. For example, if your project name is myproject:

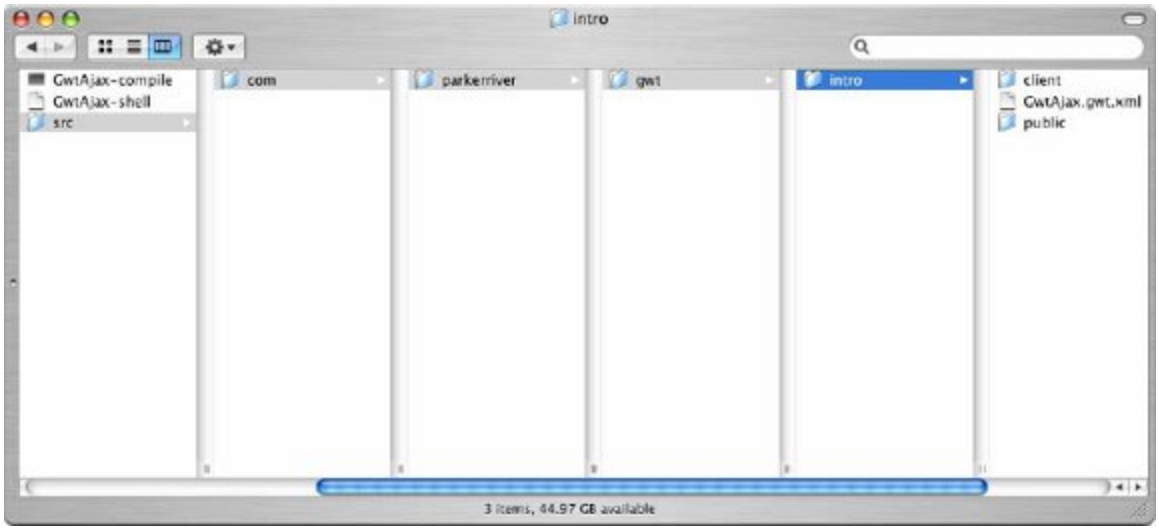
```
projectCreator -ant myproject -out /users/bruceperry/1gwt/test -overwrite
```

this command creates an Ant file that will compile the Java files in */users/bruceperry/1gwt/src* into */users/bruceperry/1gwt/bin*. The command also creates the */users/bruceperry/1gwt/src* and */users/bruceperry/1gwt/test* directories.

For more details, see the documentation at:
http://code.google.com/webtoolkit/documentation/.projectCreator.html

Figure 2 shows the folders and files that `applicationCreator` generated.

Figure 3-2. Application setup for a GWT project



# Chapter 4. The GWT's Directory Structure

Whether you use the `applicationCreator` script or not, this is what your directory structure should look like. The client directory is where the Java class representing your user interface resides (such as the entry point class), and any of this class's support classes.

The public directory contains an HTML file that has the same name as your client class, but with a different suffix: *GwtAjax.html*. This file is only necessary if you are not dynamically generating your entire user interface with JavaScript. The HTML file allows you to enclose the JavaScript associated with your application widgets within a bigger design. This is typically the way you will go, as the designers will be building the XHTML pages, which will point to the GWT-generated JavaScript. The dynamically generated user interface aspects are usually enclosed by XHTML `div` tags in these pages.

# Chapter 5. Modules

The script also generated a basic module file: *GwtAjax.gwt.xml*. This is a required configuration file for your application. The module XML file specifies any entry-point class for your application, other modules from which your application inherits, and additional settings that you may use less often. The code sample shows the basic module that `applicationCreator` generates. Later, this shortcut shows the changes we make to this file to inherit additional modules and implement internationalization (i18n).

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

  <!-- Specify the app entry point class.                   -->
     <entry-point class='com.parkerriver.gwt.intro.client.GwtAjax'/>
</module>
```

This file indicates that the module inherits from an existing module, `com.google.gwt.user.User`, which the GWT provides. The file also specifies the fully qualified entry-point class for the application.

> NOTE
>
> *Internationalization* is a popular term for allowing your application to handle different languages. It is also known as i18n (the word begins with an *i*, is followed by 18 letters, and ends with an *n*). Most web pages will be viewed by people from all over the world. You can specify that the GWT application handles certain languages by using the techniques explained elsewhere in this shortcut. For example, our test application will display messages in English, Spanish, and German.

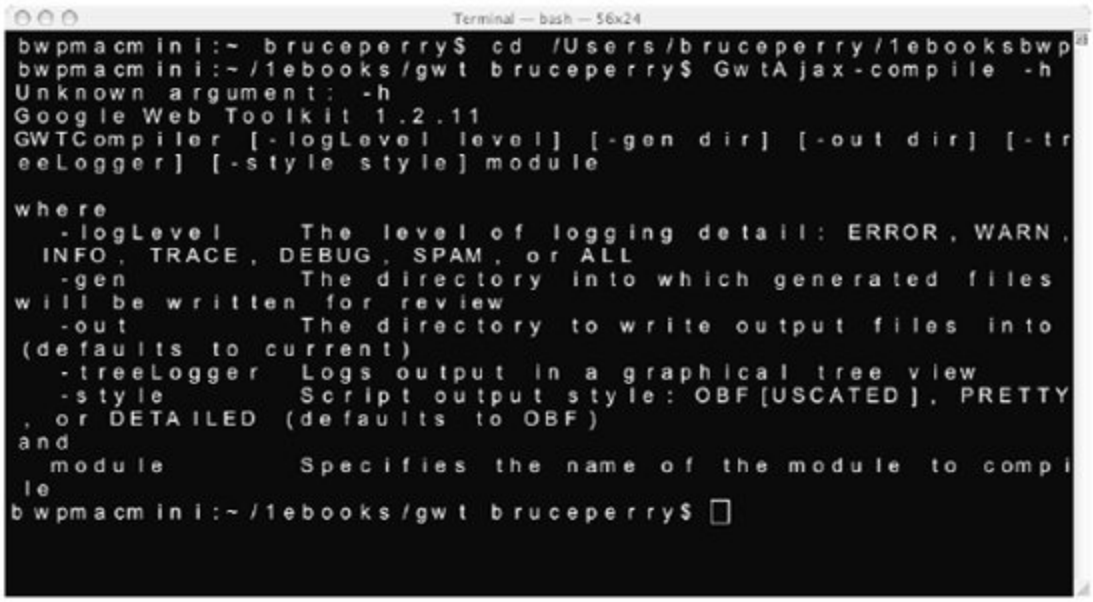
# Chapter 6. GWT Compiler and Web Mode

Two other files in the generated directory in Figure 2 jump out at us: the `GwtAjax-Compile` and `GwtAjax-shell` scripts. You can run these scripts from the command line, or dynamically from an Ant file. If your application was called XXX, then `applicationCreator` would call these files *XXX-Compile* and *XXX-Shell*.

The `GwtAjax-Compile` script uses a Java class to compile your Java code into a JavaScript file (if the code doesn't contain any errors). Once you have compiled the code, you can test or view the application in the browser(s). GWT refers to the latter development method as *web mode*.

An easy way to compile your code is to enter `GwtAjax-Compile` at the command line with no options. Figure 3 displays the description of the `GwtAjax-Compile`'s various options such as `-out` or `-style`.

Figure 6-1. Running an XXX-Compile script to compile from Java code to JavaScript



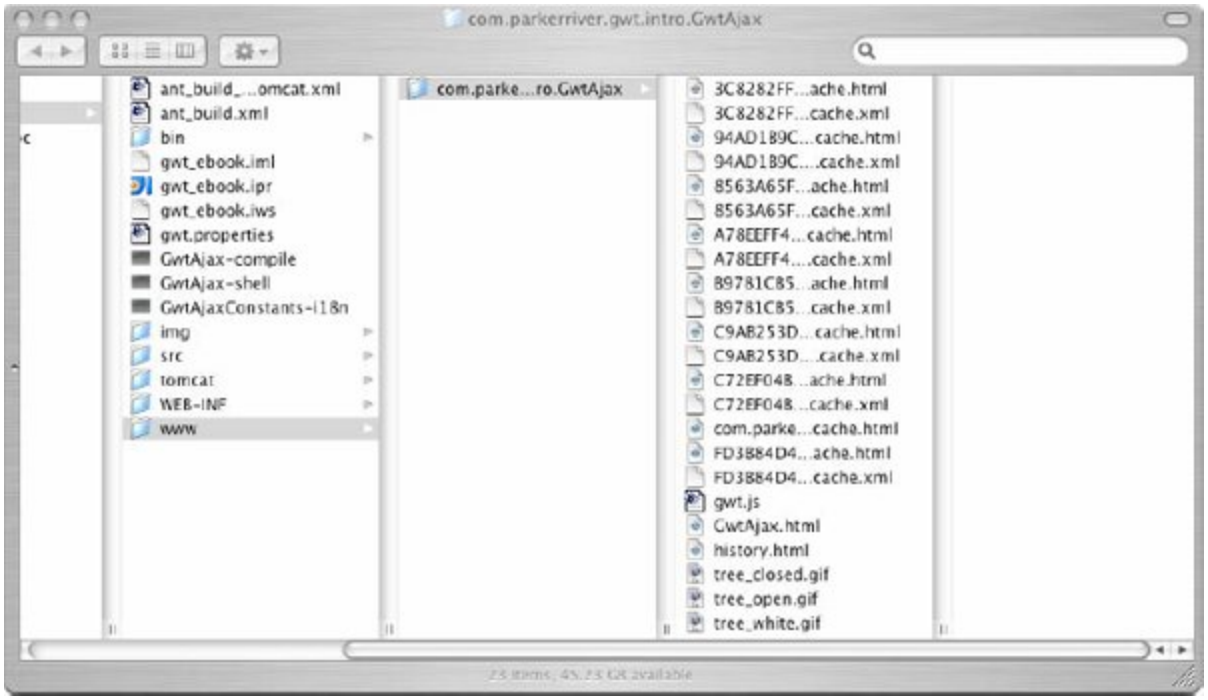

For example, if you type:

```
XXX-Compile -style PRETTY -out myapp
```

then the compiler will generate a non-obfuscated JavaScript file (where the code is easier to read for someone looking at the *gwt.js* file) inside a directory named *myapp/fully-qualified-name-of-your-application-class*. An example output directory from this shortcut's application is *www/com.parkerriver.gwt.intro.GwtAjax*. A directory named for a Java class is awkward to type into a browser, so you can change this directory name by using an Ant build file later, or by using some other method.

After compiling, what does the output directory contain? Figure 4 shows the contents of the *www/com.parkerriver.gwt.intro.GwtAjax* directory.

Figure 6-2. The contents of the XXX-compile output directory




The directory certainly contains a lot of software artifacts! The *gwt.js* file contains the generated JavaScript. This directory also contains the HTML or XHTML that your users will load into their browsers (e.g., *GwtAjax.html*). The output also includes several files of the form *C72EF04BD1BEF69CF1EC5772367A07B0.cache.html*. These files are related to browser-specific versions of the generated JavaScript code, and will be part of the files (along with *gwt.js*) that you deploy to your production server. The XML files of the form *C72EF04BD1BEF69CF1EC5772367A07B0.cache.xml* do not have to be deployed. The GWT compiler class uses these files as metadata during the compilation process.

PREV NEXT

# Chapter 7. Host Mode

What's the *GwtAjax-shell* script for? This script is another one of the GWT's command-line tools. The script will launch a special browser that the GWT provides, as well as a window that will display a log with debugging information for your application. This special browser is designed to debug and display your application as Java objects loaded into a Java Virtual Machine (JVM), before the GWT has generated JavaScript from it.

Along with the coding of your application in an IDE, the GWT calls this development stage *host mode* . Figure 5 shows a GWT application running in host mode.

Here is what the `GwtAjax-shell` script or `XXX-shell` looks like:

```
#!/bin/sh
APPDIR='dirname $0';
java -XstartOnFirstThread -cp "$APPDIR/src:$APPDIR/bin:/Users/bruceperry/1gwt/gwt-mac-
1.2.11/gwt-user.jar:/Users/bruceperry/1gwt/gwt-mac-1.2.11/gwt-dev-mac.jar"
com.google.gwt.dev.GWTShell -out "$APPDIR/www" "$@"
com.parkerriver.gwt.intro.GwtAjax/GwtAjax.html;
```

Notice the Java classpath, the quoted string following the `-cp` option in the shell script. It includes the */src* and */bin* directories in the top level of the application, as well as the *gwt-user.jar* and *gwt-dev-mac.jar* (or, for instance, *gwt-dev-linux.jar* ) libraries.

The code that you use or depend on to create your application, before launching host mode, must be included on the classpath. For instance, if you are using various JAR files, such as *jdom.jar, junit.jar* , or *log4j.jar* (representing the JDOM XML-handling library, the JUnit unit-testing framework, and the log4j logging classes, respectively) to develop server-side classes, and these classes are a part of the GWT application, then include the appropriate JARs in this shell script's classpath.

Figure 7-1. A GWT application running within the framework's special browser


The purpose of this browser mode is to allow the developer to make incremental changes in the application, then get visual feedback, before the tool compiles the widgets and underlying logic into JavaScript. You can also compile your Java code into JavaScript within the host mode environment, which then launches your default browser, such as Firefox, to display the application. To accomplish this task, click the Compile/Browse button on the browser.

PREV NEXT

# Chapter 8. Demo Application

Phew, now we have plowed through a lot of the details dealing with the GWT development environment. What does our application involve? We'll keep it simple but real-world, not just your garden variety "Hello World" application. Figure 6 shows the screen for the application. The user clicks the "Request Info" button, and the three text fields are filled with the server name, the time of the user's request, and the associated user-agent string (a special identifier that is associated with each browser vendor).

Figure 8-1. The screen for our demo application in Firefox


Obviously, all this information is derived from a server componentin our case, a PHP file. We also have developed a Java servlet for the same purpose, which the shortcut will show in a later section.

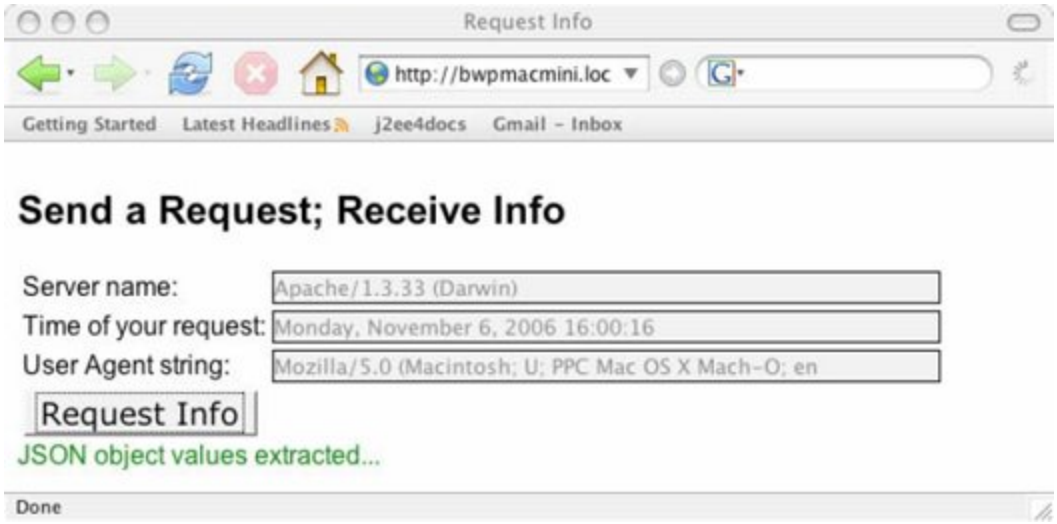
The application user does not experience a browser refresh; the information appears in the text boxes after a few moments. Messages appear beneath the form widgets, letting the user know the status of her button click.

Figure 7 shows the screen after the user first clicks the button. An alert box shows the server return value (useful for debugging). A message in a large, green font indicates that the form values have been submitted. This message may seem obvious, but the user is accustomed to seeing the browser page refresh or reconstitute when she submits web form values. In Ajax applications, nothing happens upon this submission; the browser sends the data behind the scenes. Therefore, a message provides the user with a visual cue about the application's progress.

Figure 8-2. The user views HTTP request information in an Ajaxy manner


Figure 8 shows the final stage of handling the server return value, after the user has dismissed the alert window. The text boxes contain the server information.

Figure 8-3. Magic JSON provides information from the server




The message area displays a new message about JSON values being extracted. The server has returned its information in JSON format. The client side of the application extracts the data from this return value. An upcoming section about the application's source code explains this mechanism in more detail.

Now we will look at what is happening on the server side.

PREV NEXT

# 8.1. The Server Side

Our GWT application connects to a PHP file running on the Apache web server. This PHP file provides the information from the server, which ends up displayed in the text fields. Here's the source code for the PHP:

```
<?php
//set the content-type HTTP header
header("Content-Type: text/plain; charset=UTF-8");
//Indicate that the browser should not cache the received value
header("Cache-Control: no-cache");
$info_array = array(
"server" => $_SERVER["SERVER_SOFTWARE"],
"date" => date("l, F j, Y H:i:s",$_SERVER["REQUEST_TIME"]),
"browser" => $_SERVER["HTTP_USER_AGENT"]);
echo json_encode($info_array);
?>
```

This code sets a couple of HTTP response headers that specify the return value's content type (`text/plain; charset=UTF-8`), and that the browser, or any other caching mechanism along the request-response chain, should not cache the received value. Then the code creates an array of data to send about the server software, the request's date, and the user-agent header value.

The program uses the `$_SERVER` global variable to obtain its information. For example, you can access the user agent string with the code: `$_SERVER["HTTP_USER_AGENT"]`.

Finally, the PHP uses a method named `json_encode()` to encode the array in JSON format, and the `echo` statement to print the HTTP response.

If you look at Figure 7 and the `alert` window, you will see exactly what the format of the returned information looks like. JSON is an easy-to-use format to exchange between the client and server tiers of an Ajax application.

> NOTE
>
> An Ajax application that handles JSON return values typically converts these values to JavaScript objects using, among other techniques, JavaScript's built-in `eval()` method. This design strategy, however, can potentially open up an application to security vulnerabilities such as cross-site scripting (XSS) attacks. See http://en.wikipedia.org/wiki/Cross_site_scripting.

For example, since the JSON response is interpreted as JavaScript, a hacker could intercept an HTTP response and inject annoying (pop-up windows with bizarre messages) or malicious program code into the application. Developers are encouraged to keep this security implication in mind when using JSON as a data format. One defensive strategy is to use regular expressions or some other mechanism to filter the response text before the application converts the JSON-formatted `string` to a JavaScript object. This is admittedly an imperfect solution, as it couples the server component to the client by specifying in the client a certain format for an HTTP response.

Of course, substituting XML for JSON is not a completely bulletproof strategy either, as the HTTP response could be well-formed XML containing distorted or malicious element content. For this reason, both server-side and client-side developers have to give security the highest priority when designing Ajax applications.

An alternative to PHP is the following Java servlet. The advantage of a servlet is that it can easily be integrated with GWT's embedded instance of the Tomcat servlet container, when you are using GWT in host mode.

```
package com.parkerriver;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;

import org.json.JSONObject;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletRequest;

public class AjaxEbookServlet extends HttpServlet {

protected void doGet(HttpServletRequest httpServletRequest,
                     HttpServletResponse httpServletResponse) throws
ServletException, IOException {
  //set Content-Type and Caching headers
  httpServletResponse.setHeader("Content-Type",
      "text/plain; charset=UTF-8");
  httpServletResponse.setHeader("Cache-Control",
      "no-cache");
  //for backward compatibility with HTTP/1.0...
  httpServletResponse.setHeader("Pragma",
      "no-cache");
  Map infoMap = new HashMap();
  infoMap.put("server",
      httpServletRequest.getServerName());
  infoMap.put("date",new java.util.Date().toString());
  infoMap.put("browser",httpServletRequest.
    getHeader("user-agent"));
  //The object that writes to the response stream
```

```
  PrintWriter writer = httpServletResponse.getWriter();
  //See: http://www.json.org/java/index.html
  JSONObject jObj = new JSONObject(infoMap);
  writer.print(jObj.toString());
  writer.close();

}

protected void doPost(HttpServletRequest httpServletRequest,
 HttpServletResponse httpServletResponse) throws ServletException, IOException {
  doGet(httpServletRequest, httpServletResponse);
 }
}
```

Now we will examine the guts of it all: the HTML file, *GwtAjax.html* and the Java file that forms the application's logic, *GwtAjax.java*.

PREV NEXT

# 8.2. HTML File

I made quite a few small changes to the HTML file that `applicationCreator` automatically generated. This is mainly because, as in many real-world applications, the HTML should be valid against a certain markup type. I changed the HTML so that the markup conforms to XHTML Transitional. I used the World Wide Web Consortium's markup validation service to check the file: http://validator.w3.org.

Here is the XHTML file's source code.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang=
      "en" lang="en">
<head>
  <meta http-equiv="content-type" content=
         "text/html; charset=UTF-8" />
  <link rel="stylesheet" type="text/css" href="gwttest.css" />
  <title>Request Info</title>
  <!--                                              -->
  <!-- The module reference below is the link       -->
  <!-- between html and your Web Toolkit module     -->
  <!--                                              -->
  <meta name='gwt:module' content=
         'com.parkerriver.gwt.intro.GwtAjax' />
</head>
<body>

<!--                                            -->
<!-- This script is required bootstrap stuff.   -->
<!-- You can put it in the HEAD, but startup    -->
<!-- is slightly faster if you include it here. -->
<!--                                            -->
<script type="text/javascript" language=
      "javascript" src="gwt.js"></script>

<!-- OPTIONAL: include this if you want history support
           <iframe id="__gwt_historyFrame"
style="width:0;height:0;border:0"></iframe> -->

<h2>Send a Request; Receive Info</h2>
<!The form will dynamically be generated inside this div -->
<div id="container"></div>
<!This div contains the status messages-->
<div id="status"></div>
<!This div displays any error messages-->
<div id="err_message"></div>
```

```
</body>
</html>
```

The changes I made in the HTML included the DOCTYPE at the top; the addition of a "type="text/javascript"" attribute to the `script` tag, properly closing the `meta` tags, and the like. The HTML file includes `div` tags at the bottom. The JavaScript that the GWT generates for this application, from the Java source in *GwtAjax.java*, will dynamically write the `form` tags and any messages inside these tags, as well as handle the button-click event.

One `div` will contain the form elements, and the next two `divs` will contain informational and any error messages.

Now we can finally look at the Java source for our client.

PREV NEXT

# 8.3. Java Source

This is the Java source code that creates all of the visible widgets, except for one `h2` tag, for our application. The code represents a Java file like most others: a package statement followed by the importation of several classes, which are parts of the GWT API, followed by a class definition.

Like a Java file that uses Swing, this file includes a lot of GUI-related code. The source code is commented, but you might look at Figure 7 or 8 to recall what the window looks like in order to help you understand what is going on here.

```
package com.parkerriver.gwt.intro.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.HistoryListener;
import com.google.gwt.user.client.DOM;
import com.google.gwt.user.client.Element;
import com.google.gwt.user.client.HTTPRequest;
import com.google.gwt.user.client.ResponseTextHandler;
import com.google.gwt.user.client.ui.Widget;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.Grid;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.FormPanel;
import com.google.gwt.user.client.ui.RootPanel;

import com.google.gwt.json.client.JSONObject;
import com.google.gwt.json.client.JSONParser;
import com.google.gwt.json.client.JSONValue;
import com.google.gwt.json.client.JSONException;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class GwtAjax implements EntryPoint,
    HistoryListener {
  //The URL that the HTTPRequest.asyncGet() method uses
  public static final String TARGET_URL =
      "http://bwpmacmini.local/~bruceperry/info2.php";
  private final FormPanel form = new FormPanel();
  //A Grid, or HTML table, with 5 rows and 3 columns
  private final Grid formGrid = new Grid(5, 3);
  //The server name goes in this read-only TextBox
  private final TextBox servBox =
```

```
    createTextBox(50, 50, "", null);
//The request time goes in this read-only TextBox
private final TextBox timeBox =
    createTextBox(50, 50, "", null);
//The browser's user-agent string fills this TextBox
private final TextBox browserBox =
    createTextBox(50, 50, "", null);
//Check the return value against this regular expression
private final static String regexCheck =
  "^\\{\"(server|browser|date)\":\".+\"\\,\"(server|browser|date)\":"+
"\".+\"\\,\"(server|browser|date)\":\".+\"\\}$";



/**
 * This is the entry point method, called when
 * the user loads the application into the browser.
 */
public void onModuleLoad() {

  final Label servLab = createLabel("Server name: ");
  //The textboxes are not editable;
  // they just display values
  servBox.setEnabled(false);
  //Add these widgets to the Grid's first row
  formGrid.setWidget(0, 0, servLab);
  formGrid.setWidget(0, 1, servBox);
  final Label timeLab = createLabel(
      "Time of your request: ");
  timeBox.setEnabled(false);
  //Add the request-time label and TextBox widgets
  //to the Grid's second row
  formGrid.setWidget(1, 0, timeLab);
  formGrid.setWidget(1, 1, timeBox);
  final Label browserLab = createLabel(
      "User Agent string: ");
  browserBox.setEnabled(false);
  //Add the user-agent label/TextBox to the Grid's third row
  formGrid.setWidget(2, 0, browserLab);
  formGrid.setWidget(2, 1, browserBox);
  //Finally, add the submit button
  Button but = new Button("Request Info");
  //The GwtIntroListener is a nested class; see below
  but.addClickListener(new GwtIntroListener());
  //Position the button in the grid's last row
  formGrid.setWidget(3, 0, but);
  //Add the Grid to the form
  form.setWidget(formGrid);
  //Add the entire form/grid widget to a div
  // with id "container"
  RootPanel root = RootPanel.get("container");
  if (root != null) root.add(form);
```

```
}

 /* A convenience method for creating labels */
 protected Label createLabel(String labText) {
   Label lab = new Label();
   lab.setText(labText);
   return lab;
 }

 /* A convenience method for creating TextBoxes */
 protected TextBox createTextBox(int len, int visibleLen, String name,
                                 String width) {
   TextBox tb = new TextBox();
   tb.setMaxLength(len);
   tb.setVisibleLength(visibleLen);
   if (! (name == null || name.equals(""))) tb.setName(name);
   //the object's new width, in CSS units (e.g. "10px", "1em")
   if (! (width == null || width.equals(""))) tb.setWidth(width);
   return tb;
 }

 public void onHistoryChanged(String string) {
   RpcStatus stat = new RpcStatus();
   stat.setStatusDivId("err_message");
   stat.showStatus(true, "token: "+string, "purple");
 }

 private boolean checkResponseValue(String val) {
   return val.matches(regexCheck);
 }

 private class GwtIntroListener implements
     ClickListener{

   public void onClick(Widget widg) {
     //Use this object to show the status of the request
     final RpcStatus status = new RpcStatus();
     //Allow different translations of these messages
     //by using GWT's i18n interfaces
     final GwtAjaxConstants myConstants =
         (GwtAjaxConstants) GWT.
             create(GwtAjaxConstants.class);
     status.showStatus(true,
         myConstants.formSubmissionMsg(), "green");
     //Send the asynchronous request for the server info.
     //Use the com.google.gwt.http.client package for
     //more functionality.
     HTTPRequest.asyncGet(
         TARGET_URL,
         new ResponseTextHandler() {
           public void onCompletion(
               String responseText) {
```

```
            //debug return value...
            //Window.alert(responseText);
            if (responseText == null ||
                responseText.equalsIgnoreCase("")) {
              status.setStatusDivId("err_message");
              status.showStatus(true,
                  "The server appears to be unavailable right now." +
                      " Please try again in a moment.",
                  "red");
              return;
            }
            try {
              //Check the return value against our regular expression
              if (! checkResponseValue(responseText))
                throw new JSONException(
                    "The return value was invalid.");
              JSONValue jval = JSONParser.parse(
                  responseText);
              JSONObject jobj = jval.isObject();
              if (jobj != null) {
                //Write the info into the textboxes
                servBox.setText(jobj.get("server").
                    isString().stringValue());
                timeBox.setText(jobj.get("date").
                    isString().stringValue());
                browserBox.setText(jobj.get("browser").
                    isString().stringValue());
                status.showStatus(true,
                    myConstants.JsonMsg(), "green");
              }
            } catch (JSONException je) {
              status.setStatusDivId("err_message");
              status.showStatus(true, "Exception message: " +
                  je.getMessage(), "red");
            }

          }
        });//end asyncGet
  }//end onclick
}//end ClickListener nested class
//Various Getters for using in JUnit tests
public TextBox getServBox() {
  return servBox;
}

public TextBox getTimeBox() {
  return timeBox;
}

public TextBox getBrowserBox() {
  return browserBox;
}
```

```
  public FormPanel getForm() {
    return form;
  }

  public Grid getFormGrid() {
    return formGrid;
  }
}
```

PREV NEXT

# 8.4. Using a Subset of the J2SE

For client-side Ajax coding in Java, the GWT only supports certain classes from the J2SE `java.lang` and `java.util` packages. Recall that your Java code is destined to be output as JavaScript, so the GWT API has to emulate each of the Java classes, such as `ArrayList, Integer`, or `String`, in JavaScript.

For example, I started using the `java.text.DateFormat` class, until I remembered that the `java.text` package is not available (right now) in the GWT API. So for that particular application, I had to rely on the `java.util.Date` functionality in lieu of creating my own module for the `java.text` classes.

Many of the classes that Java programmers use quite often from `java.lang`, however, are mirrored in the GWT. The GWT has included `ArrayList, HashMap`, and `HashSet`, for instance. Typically, you will find that enough GWT API classes exist to help you accomplish your task.

The following API web pages describe which classes the GWT makes available in `java.lang` and `java.util`.

java.util: http://code.google.com/webtoolkit/documentation/java.util.html

java.lang: http://code.google.com/webtoolkit/documentation/java.lang.html

### NOTE

You can use the breadth of J2SE and JEE packages in the server-side classes that are linked to your GWT application. This is because those objects will run within a Java Virtual Machine (JVM) on the server. For example, imagine that you use the GWT's RPC service API, in which a JavaScript object makes a remote procedure call to a servlet and exchanges data with that component. The servlet could use the `java.text` package, but the client-side classes that interact with it cannot, because they are restricted to the `java.lang` and `java.util` packages.

# 8.5. GUI Classes

The main purpose of the `onModuleLoad()` method is to build the GUI and handle the button's click event. Most of the GUI classes derive from the `com.google.gwt.user.client.ui` package, a part of the GWT API.

For example, the code uses a `FormPanel` and places a `Grid` or HTML table inside the form in order to position the `Labels, TextBoxes`, and `Button`.

> NOTE
>
> The code instantiates a few of the GUI elements, a `FormPanel` and a `Grid`, outside of the `onModuleLoad()` method. In other words, they are instance variables rather than local variables. As a result, a JUnit test-case class can use a getter method to access the `Grid`, for instance, and test the state of the widgets inside of it.

The `TextBoxes` cannot be edited by the user, since they only have a display purpose (`TextBox.setEnabled(false)`).

# 8.6. Cascading Style Sheets

Figure 8 indicates that the `TextBoxes` have a thin border around them, and the `Button's` label uses a different font than other elements on the page. (OK, so a professional designer might frown at these choices' lack of aesthetic quality!) The CSS file that is linked to the HTML controls these visual aspects of the application. Here is the link tag in the HTML that connects to our CSS file.

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

Your code can automatically reference GUI elements using the syntax `gwt-Button` or `gwt-TextBox` in the CSS file selectors, as in our file *gwttest.css*.

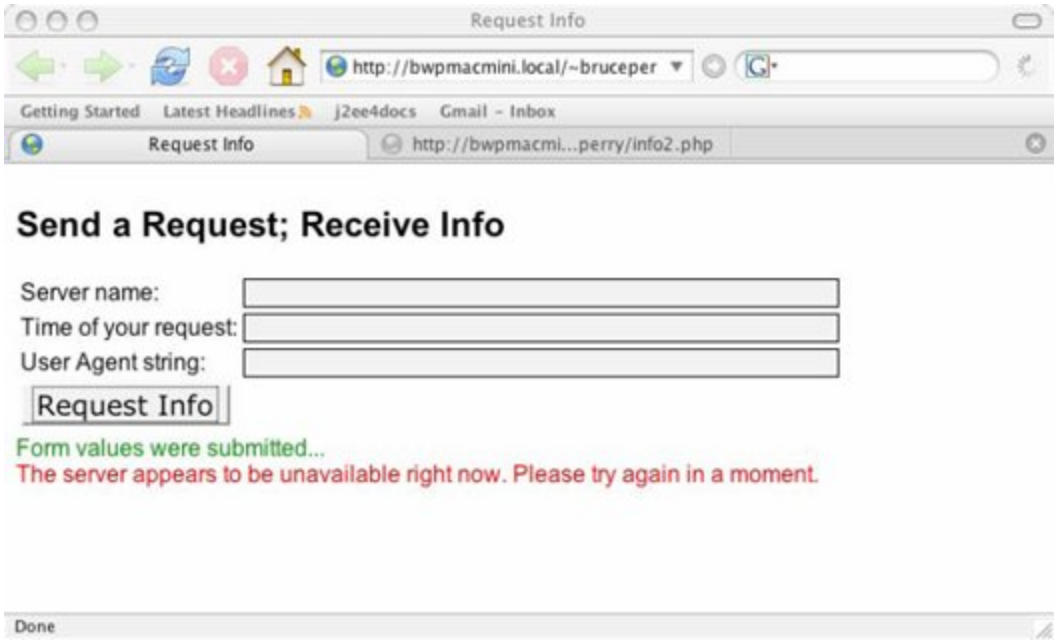
```
.gwt-TextBox { border: thin solid black}
.gwt-Button { font-family: Verdana, serif; font-size: 1.2em}
```

You can also dynamically change the style attributes of a GUI element in Java code, as in the upcoming `RpcStatus` class.

# 8.7. Displaying User Messages

I created a utility class called `RpcStatus`, which is used to show messages inside one of two `div` tags. One `div` is designed for informational messages, the other for errors or exceptions. Figure 9 shows an informational message and an error message.

Figure 8-4. An Ajax application shows an error message because the server is down




Here is the source code for that utility class.

```
package com.parkerriver.gwt.intro.client;

import com.google.gwt.user.client.Element;
import com.google.gwt.user.client.DOM;

/**
 * A utility class for displaying messages
 */
public class RpcStatus {
  private String statusDivId;
```

```
  public RpcStatus() {
      this("status");
  }

  public RpcStatus(String statusDivId) {
      this.statusDivId = statusDivId;
  }

  public String getStatusDivId() {
      return statusDivId;
  }

  public void setStatusDivId(String statusDivId) {
      this.statusDivId = statusDivId;
  }

  public void showStatus(boolean _on, String message,
                         String color){
      Element el = DOM.getElementById(getStatusDivId());
      if(el != null)  {
          if(_on) {
              DOM.setStyleAttribute(el,"font-size","1.2em");
              if(color == null || color.length() < 1) {
                  DOM.setStyleAttribute(el,"color","purple");
              } else {
                  DOM.setStyleAttribute(el,"color",color);
              }
              if(message == null || message.equalsIgnoreCase("")){
                  DOM.setInnerHTML(el,
                          "");
              } else {
                  DOM.setInnerHTML(el, message);
              }
          }  else{
              DOM.setInnerHTML(el, "");
          }
      }

  }
}
```

One interesting aspect of this class is that it uses the `com.google.gwt.user.client.DOM` object to implement the kind of classic Ajaxy programming we are all fond of, using familiar DOM API methods such as `getElementById()`, and techniques such as setting an element's `innerHTML` property (using the method `DOM.setInnerHTML()`).

The source code for displaying the error message in Figure 9 is:

```
status.setStatusDivId("err_message");
status.showStatus(true,
"The server appears to be unavailable right now."+
" Please try again iin a moment.",
"red");
```

PREV NEXT

# 8.8. Inheriting Other Modules

I changed the module XML file for this application in order to use the JSON- and i18n-related modules that the GWT provides. The reason? I used the JSON module to handle the JSON return value from the PHP file. In addition, I used an interface from a module called `com.google.gwt.i18n.I18N`, so that the application can optionally display its messages in Spanish or German.

Remember that all of the Java classes that you are writing for the client side of the Ajax application will eventually be converted to their corresponding JavaScript. Therefore, when your project's module inherits another module, what you are really doing is importing the JavaScript that other developers have created. The GWT provides modules that handle the JSON format and i18n programming, among others. Here is what our altered module looks like (in `GwtAjax.gwt.xml`):

```
<module>

  <!-- Inherit the core Web Toolkit stuff.                  -->
  <inherits name='com.google.gwt.user.User'/>
  <inherits name="com.google.gwt.json.JSON"/>
  <inherits name="com.google.gwt.i18n.I18N"/>
  <extend-property name="locale" values="es"/>
  <extend-property name="locale" values="de"/>
  <extend-property name="locale" values="es_ES"/>
  <extend-property name="locale" values="de_DE"/>

  <!-- Specify the app entry point class.                   -->
  <entry-point class='com.parkerriver.gwt.intro.client.GwtAjax'/>

</module>
```

Inheriting `com.google.gwt.user.User` is standard for most GWT applications. If you want to use the JSON and i18n classes, then include the additional XML elements that the prior code sample shows.

What are the `extend-property` elements? These are elements indicating that the application will handle two locales other than the default `en_US`. A locale designates the language that a web page will use, as well as formatting techniques such as for monetary values used in specific countries or regions.

A locale is made up of a language code, optionally followed by an underscore character and a country code. So `en_US` refers to English as spoken in the United States; `de_DE` represents German as spoken in Germany (remember, other countries such as Switzerland also speak German). An upcoming section describes how to handle these locales in your application. First I will show the JSON-related code.

# 8.9. Using the JSON Module

Our application's module inherits the GWT's JSON module. Our `GwtAjax.java` class imports classes that are part of this API. If you do not include the associated module in your own module XML file, as in:

```
<inherits name="com.google.gwt.json.JSON"/>
```

then the GWT compiler will complain, for example, that it cannot resolve the following import statements.

```
import com.google.gwt.json.client.JSONObject;
import com.google.gwt.json.client.JSONParser;
import com.google.gwt.json.client.JSONValue;
import com.google.gwt.json.client.JSONException;
```

We are using these classes because the client-side code, which will eventually be JavaScript, will handle a JSON return value from the server. A simplified version of the return value looks like this:

```
("server":"Apache","date":"November 8, 2006","browser":"Firefox"}
```

The client-side code pulls the server, date, and browser-related information out of this object and displays them in the `TextBoxes`. As mentioned in a previous note, evaluating JSON formats as program code can open up your application to malicious insertions of JavaScript (I hope this phrase does not evolve into another acronym: MIJ!)

Therefore, the client application uses a regular expression to filter the textual server return value before the `string` is evaluated as JavaScript code.

For example, if the server sends back code such as `alert('Gotcha!')`,then the following code piece will actually end up displaying that browser window and message:

```
JSONValue jval = JSONParser.parse(responseText);
```

Here is the `GwtAjax.java` code for accomplishing these filtering and formatting actions:

```
//The regular expression is an instance variable
private final static String regexCheck =
 "^\\{\"(server|browser|date)\":\".+\"\\,\"(server|browser|date)\":"+
 "\".+\"\\,\"(server|browser|date)\":\".+\"\\}$";
//..Later on in the code
try {
   if (! checkResponseValue(responseText))
       throw new JSONException("Return value was invalid.");
   JSONValue jval = JSONParser.parse(responseText);
   JSONObject jobj = jval.isObject();
    if (jobj != null){
        //Fill in the server info
        servBox.setText(jobj.get("server").isString().
        stringValue());
        //Fill in the date info
        timeBox.setText(jobj.get("date").isString().
        stringValue());
        //Fill in the browser info
        browserBox.setText(jobj.get("browser").isString().
        stringValue());
        //Refresh the status message
        status.showStatus(true,
        myConstants.JsonMsg(),"green");
    }
} catch (JSONException je)  {//display Exception message}

//The method that uses the regular expression
private boolean checkResponseValue(String val) {
  return val.matches(regexCheck);

}
```

This code parses the return value and pulls out the `string` values, using objects from GWT's JSON classes.

The regular expression String is quite difficult to look at; therefore, here is a narrative description of what the expression is designed to accomplish.

The String the expression checks for begins with "{" and double quotations, then continues with either the words server, browser, or date, followed by ending quotations and a colon (":"). In the pattern, these characters are followed by one or more characters in double quotations and a comma. The expression repeats the latter pattern twice until the String concludes with end doouble quotes and a "}" character.

NOTE

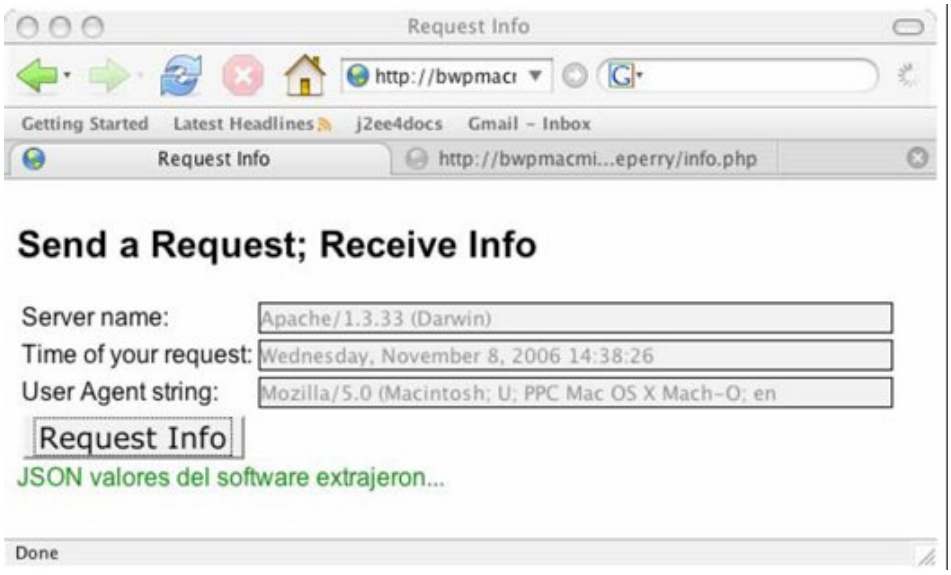
What if we wanted to handle return values formatted as XML instead of JSON? We can do that! We just need to import another module in our *XXX.gwt.XML* file. This module is named `com.google.gwt.xml.XML`. Inherit it the same way that your module inherited the JSON moduleby adding an `inherits` element to your own module. Then you use the GWT API classes in the `com.google.gwt.xml.client` package to handle the HTTP responses in XML format.

PREV NEXT

# 8.10. Accommodating Different Languages

Most web sites make an effort to display their messages in different languages, or are simply required to because of the nature of their business. Figure 10 shows an informational message for a user with a Spa locale specified. Pardon the translation if it does not make sense or unintentionally says something silly o worse, as I am not a professional translator (I used Google's language tool)! We're just using these translations as an example of how to implement the GWT's i18n mechanism.

Figure 8-5. An informational message in Spanish




If you are familiar with Java's mechanism involving ResourceBundles and property files, then you will recognize GWT's i18n methods. First, we create a default properties file named *GwtAjaxConstants.properties* to handle our messages. The application stores this file in the same directo as the *GetAjax.java* file: *com.parkerriver.gwt.intro.client* .

```
#sample constant property to be translated in language specific versions of this
#property file
formSubmissionMsg=Form values were submitted...
JsonMsg=JSON object values extracted...
```

I specified the Spanish version of these messages in the file *GwtAjaxConstants_es_ES.properties* . Java developers use this file-naming convention for property files that represent certain locales. This one represents the "Spanish speakers in Spain" locale, which is specified by `es_ES` .

```
formSubmissionMsg=Los valores de la forma fueron sometidos
JsonMsg=JSON valores del software extrajeron...
```

We also included a properties file for German speakers, which we do not have to show. That file name is *GwtAjaxConstants_de_DE.properties* . "de" is the language code for German; "DE" is the country code fo Germany.

Next, you have to provide an interface that extends `com.google.gwt.i18n.client.Constants` .

> NOTE
>
> This is only one i18n mechanism that is included in the GWT framework. For a description of the others, see the documentation at:
>
> *http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.*
>
> *DeveloperGuide.Internationalization.html*

This interface defines the methods your code will call to dynamically generate messages for specific locale How the code accomplishes this task will be more apparent when I show the specific locale-related code i the `GwtAjax.java` class. Here is the code for the `GwtAjaxConstants.java` interface.

```
package com.parkerriver.gwt.intro.client;
/**
 * Interface to represent the constants contained in resource bundle:
 *  /gwt/src/com/parkerriver/gwt/intro/client/GwtAjaxConstants.properties'.
 */
public interface GwtAjaxConstants extends
        com.google.gwt.i18n.client.Constants  {

    /**
     * Translated "Form values were submitted...".
     *
     * @return translated "Form values were submitted..."
     */
   String formSubmissionMsg();
   /**
     * Translated "JSON object values extracted...".
     *
     * @return translated "JSON object values extracted..."
     */
  String JsonMsg();
```

```
}
```

This code defines two methods: `formSubmissionMsg()` and `JsonMsg()` . The method names line up with th
names of the properties in the properties files.

```
formSubmissionMsg=Form values were submitted...
JsonMsg=JSON object values extracted...
```

Here is the code inside the Java file *GwtAjax.java* that dynamically displays a translated message, depending on the particular locale.

```
final GwtAjaxConstants myConstants = (GwtAjaxConstants) GWT.
      create(GwtAjaxConstants.class);
status.showStatus(true,
    myConstants.formSubmissionMsg(),"green");
...
browserBox.setText(jobj.get("browser").isString().stringValue());
status.showStatus(true,
    myConstants.JsonMsg(),"green");
```

The method chain using this JSON API is a little verbose. Here is an explanation. A `JSONObject` calls `get()`
passing in the key name as a `String` . This method returns the value associated with the key as a `JSONVa`
object. The `isString()` method returns a `JSONString` , on which you call `stringValue()` to obtain the key
value.

Remember that we have to include the following values in the module XML file. These values correspond
properties files of the form *GwtAjaxConstants_de.properties* . The latter resource will translate messages
the "de" locale, representing the German language (but not further delineating the country).

```
<inherits name="com.google.gwt.i18n.I18N"/>
<extend-property name="locale" values="es"/>
<extend-property name="locale" values="de"/>
<extend-property name="locale" values="es_ES"/>
<extend-property name="locale" values="de_DE"/>
```

PREV NEXT

# 8.11. Specifying a Certain Locale for a Web Page

So how does the application find out about which locale the user is associated with? GWT provides two ways. One involves examining the URL with which the user accesses the application and determining whether the page will display English, Spanish, or German languages. The URL includes the locale in a query string.

```
http://bwpmacmini.local/~bruceperry/ajax/com.parkerriver.gwt.intro.GwtAjax/GwtAjax.html?locale=es_ES
```

The URL has one parameter/value pair appended to its end in the form *locale=es_ES* . You may have guessed that if the locale is "es," then the translations from *GwtAjaxConstants_es.properties* are used. If the locale is "es_ES," then the application uses *GwtAjaxConstants_es_ES.properties* . If the locale is "es" but the developer has not provided a resource with that suffix (even if you have one with a "es_ES" suffix!), then the application uses the default localein this case, English (the translations inside *GwtAjaxConstants.properties* ).

A typical way to implement this mechanism in a web application is to have icons representing the various countries. The user clicks on an icon, and that link determines the value of the locale parameter.

Another more static way to accomplish the same thing is to include this type of `meta` tag in the HTML file (e.g., *GwtAjax.html* ).

```
<meta name="gwt:property" content="locale=es_ES">
```

This tag embeds the locale inside a particular web page. The query string/parameter method offers a more dynamic mechanism, since you can use it with just one web page. The i18n effort described here is admittedly limited, in that we have not translated the entire application, including the `Label` and `Button` text. For example, we could easily add a translated `Button` label by including the code in a properties file:

```
buttonText=Request Info
```

then add this method definition to the `GwtConstants` interface.

```
String buttonText();
```

Finally, the Java file for the GUI client could include this code:

```
final GwtAjaxConstants myConstants = (GwtAjaxConstants) GWT.
      create(GwtAjaxConstants.class);
Button but = new Button(myConstants.buttonText());
```

PREV NEXT

# 8.12. Building the Application with Ant

Running the `XXX-shell` and `XXX-compile` command-line tools may not be the most convenient method for developing a GWT application. What if all you want to do is compile your Java files into JavaScript, copy the altered application to your deploy directory, and refresh the browser to check out the changes? The Ant XML tool allows you to automate this activity. Here is a portion of the Ant file I used to execute GWT's compiler, then copy the results to a directory specified in a properties file.

```
<target name="gwt-compile"
               description="use gwt's compiler">
       <exec executable=
              "${module.gwt.basedir}/GwtAjax-compile"
            failonerror="true"/>
       <copy todir="${web.deploy.location}">
            <fileset dir=
                  "${module.gwt.basedir}/www/"
                  excludes="**/*.cache.xml">
          </fileset>
      </copy>
</target>
```

The `${web.deploy.location}` syntax is how Ant dereferences property values. For example, `web.deploy.location` is a property name whose value points to a directory path where the GWT compiler's output will be made available for HTTP requests. This Ant file also excludes copying to the web directory the files ending in *.cache.xml* (a part of the GWT compiler's output), because these files do not have to be deployed to your web server.

# 8.13. JUnit Testing

What if you want to test your application? The GWT includes a way to test aspects of your Ajax applicatio using the well-known JUnit framework. This shortcut introduces these concepts by showing you how to s for your dynamic Ajax graphical interface.

Here are the steps you can take to set up a JUnit test in GWT.

1. Call the `junitCreator` shell script, provided by GWT when it built your application directory. Here is

   ```
   junitCreator -junit /Users/bruceper/1ebooks/gwt/WEB-INF/lib/junit.jar -out /Users/b
   com.parkerriver.gwt.intro.client.GwtTest
   ```

   This command line created a folder in the top-level directory of my project named *test* , with the sk class called *GwtTest.java* nested within that directory (meaning the script created all of the other ne beneath *test* such as *com* and *parkerriver* ). *junitCreator* also created two new shell scripts named `GwtTest-web` .

2. Write your tests inside the test-case class, which we will show in a moment.

3. Compile the test-case class.

4. Make sure that a compiled version of your client-side GWT Java class (e.g., `GwtAjax.java` ) and any are also placed with the test-case class, which will reference your GWT class.

5. Make sure that this directory containing the compiled classes, such as */bin* , is included in the classp script (e.g., `GwtTest-hosted` ).

6. Execute the `XXX-hosted` command-line script to see the results of your tests.

Here is what the `XXX-hosted` script looks like on my system.

```
#!/bin/sh
APPDIR='dirname $0';
java -XstartOnFirstThread -Dgwt.args="-out www-test" -cp
"$APPDIR/src:$APPDIR/test:$APPDIR/bin:/Users/bruceperry/1ebooks/gwt/WEB-
INF/lib/junit.jar:/Users/bruceperry/1gwt/gwt-mac-1.2.11/gwt-
user.jar:/Users/bruceperry/1gwt/gwt-mac-1.2.11/gwt-dev-mac.jar"
junit.textui.TestRunner com.parkerriver.gwt.intro.client.GwtTest "$@";
```

The following code shows the `GwtTest` test-case class. These classes must extend `com.google.gwt.junit.` from the GWT API.

```
package com.parkerriver.gwt.intro.client;

import com.google.gwt.junit.client.GWTTestCase;
import com.google.gwt.user.client.ui.*;

/**
 * GWT JUnit tests must extend GWTTestCase.
 */
public class GwtTest extends GWTTestCase {
/**
 * Must refer to a valid module that sources this class.
 */
public String getModuleName() {
  return "com.parkerriver.gwt.intro.GwtAjax";
}

public void testCreateTextBox(){
 GwtAjax gwtaj = new GwtAjax();
 assertNotNull(gwtaj);
 gwtaj.onModuleLoad();
 TextBox tb = gwtaj.createTextBox(40,40,
              "tb",null);
 assertNotNull(tb);
}

public void testGui(){
  GwtAjax gwtaj = new GwtAjax();
  assertNotNull(gwtaj);
  gwtaj.onModuleLoad();
  Grid grid = gwtaj.getFormGrid();
  assertNotNull("Grid is not null.",grid);
  Widget wid = grid.getWidget(3,0);
  assertNotNull("Widget is not null.",wid);
  Button button = (Button) wid;
  assertTrue("Button label = request info.",
              (button.getText().equalsIgnoreCase(
                      "request info")));
  //Click the button, etc.
  button.click();

 }

}
```

The test classes have a `getModuleName()` method that must return the fully qualified class name of your m
class and run the `XXX-hosted` script that `junitCreator` generated. If the test methods pass, you should se

terminal window similar to Figure 11 .

Figure 8-6. The output from running a JUnit test with GWT


PREV NEXT

# 8.14. Finally, The Back Button

The GWT provides a way to restore the functionality of the web browser's back button in an Ajax application. Typically, the state of a single-page Ajax application is not related to the browser's back button.

> NOTE
>
> A single-page application is the software version of a one-act play. Since dynamic JavaScript can create web page widgets and change appearances on the fly, the web page, in terms of the physical file that the browser loads from the server, remains the same. The user can log in, manipulate buttons, lists, text fields, and other form elements, and the screen changes accordingly. The `XMLHttpRequest` object can exchange data invisibly with remote components, but the browser never has to fetch different web page files from the server. The state of the application does change, however, as stored in JavaScript objects and arrays.

In other words, the user changes something in the Ajax application by, say, clicking a button or making a selection-list choice, and he cannot go back to the application's previous state by clicking the browser back button. All he can do is reload the web page and thereby restore the page's initial state.

This is because the Ajax application's state is not associated with a URL in the browser's history list, as it is with conventional web browsing. In the conventional manner, if you surf from the *New York Times* home page to another web page, for example, the back button will bring the user back to the newspaper's home page. This is not true with the various states of an Ajax application, unless you use GWT's mechanism, which this shortcut introduces for you.

# 8.15. Teaching History

The GWT uses an `iframe` tag on the application's HTML page, along with the `com.google.gwt.user.client.History` class and `com.google.gwt.user.client.HistoryListener` interface, to provide back-button functionality. This shortcut provides the basics for setting this up in your Ajax application. Here are the steps.

First, make sure the `iframe` tag is uncommented in your HTML file. As initially generated by the `applicationCreator` script, as we used in this shortcut, the tag looks like this:

```
<!-- OPTIONAL: include this if you want history support -->
            <iframe id="__gwt_historyFrame"
style="width:0;height:0;border:0"></iframe>
```

> NOTE
>
> For a brief explanation of how the `iframe` tag is linked with new history items, see Mark Pruett's Hack #68 in O'Reilly's *Ajax Hacks* .

Then, create a class that implements the `com.google.gwt.user.client.HistoryListener` interface, or just have your entry-point class implement the interface. This means that any class implementing the interface, such as my `GwtAjax.java` class, must define the `onHistoryChanged(String string)` method.

For example, the code in the demo application can initialize the history or back-button mechanism in the `GwtAjax.java` class' `onModuleLoad()` method, as GWT's documentation suggests. Here is the code.

```
//Set up History or back button functionality
//History.getToken() returns any token appended to the
//page's URL following a hash mark, as in /GwtAjax.html#mytoken
String initialState = History.getToken();
if (initialState.length() == 0)
  initialState = "initialState";

// Call onHistoryChanged() to reflect the initial state,
//Since the GWT does not automatically call this method
//when the browser first loads the Ajax application.
onHistoryChanged(initialState);
// Make this class a history listener
History.addHistoryListener(this);
```

Our application includes code that adds new items to the browser's list of visited URLs or "history stack." The `History.newItem(String item)` method adds a URL to the history stack, thus causing the browser to enable the back button.

```
History.newItem(new  Date(timeBox.getText()).getTime()+"");
```

The `newItem()` method appends a hash mark, or *fragment identifier* , and a token to the end of the current URL. So the method call `History.newItem("nextState")` would generate a URL such as:

```
http://bwpmacmini.local/~bruceperry/ajax/com.parkerriver.gwt.intro.GwtAjax/GwtAjax.html#nextState
```

The `newItem()` method also causes the application to call `onHistoryChanged()` .

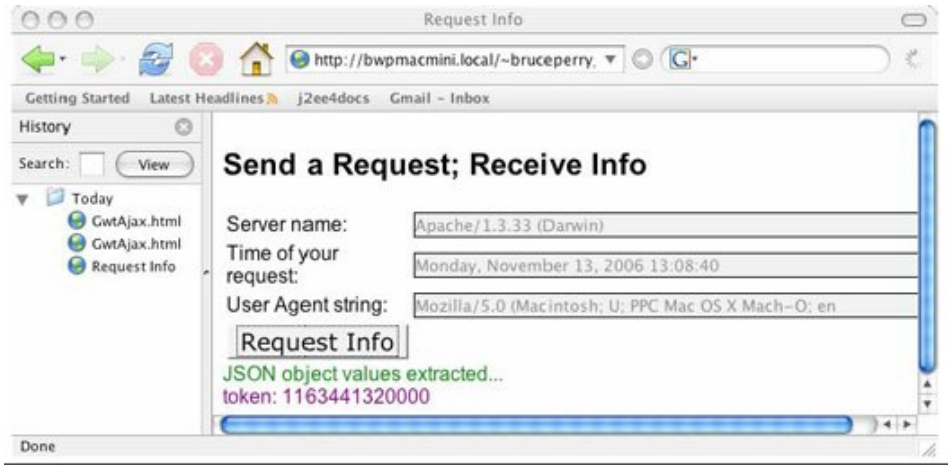
The code distinguishes between the different URLs stored in the history stack by examining the associated token (the hash mark and string at the end of the URL). The string passed into the `onHistoryChanged()` method represents the current token. Therefore, when the user clicks the back button, your implementation of `onHistoryChanged()` can examine the token and then, for instance, recreate the state associated with that segment of the history list. All the demo application does is display the current token in a purple label on the page. (See Figure 12 .)

```
public void onHistoryChanged(String string) {
 RpcStatus stat = new RpcStatus();
 stat.setStatusDivId("err_message");
 stat.showStatus(true, "token: "+string, "purple");
}
```

The application generates a unique token by taking the value of a text field that displays a date and time, and converting that date to a number by calling the `java.util.Date.getTime()` function. Figure 12 shows what the screen looks like in Firefox, including the frame showing a new list of history items, after the user has created a few new items.

```
History.newItem(new  Date(timeBox.getText()).getTime()+"");
```

Figure 8-7. Navigating with a back button through a GWT application




Each time the user clicks the "Request Info" button, the code displays the fetched server data in the text fields, creates a new token by using the second field's content, calls `History.newItem()` with the new token as an argument, and finally, displays the new value in the purple label. The result is that a new URL with the token appended at the end of it (i.e., #1163441320000) is added to the top of the history stack.

The nice thing is that GWT pays attention to when the user clicks the back or forward button, calling your `onHistoryChanged()` method accordingly.

The demo application's history-related code is, well, a demonstration. A real-world application probably wants to recreate at least a part of the application's state if the user chooses a new item on the history list. For example, your application could use the token as a key in a `HashMap` . Each `HashMap` value represents the cached state linked to the key, such as `TextBox` or `TextArea` content. Whenever the user clicks the back or forward button to reproduce a saved state, the application could check the token, get the associated value in the `Map` , then restore the `TextBox's` or `TextArea's` content.

PREV NEXT

# Chapter 9. Conclusion

This concludes the condensed tour of many of the principle tasks you can accomplish with the Google Web Toolkit. Of course, this shortcut has not covered every aspect of the GWT. For a lot more, visit the GWT documentation at http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.DeveloperGuide.html.

PREV