

Unedited
Draft



GWT IN ACTION

Easy Ajax with the
Google Web Toolkit

Robert Hanson
Adam Tacy

 MANNING



**MEAP Edition
Manning Early Access Program**

Copyright 2006 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=286>

Contents

PART I: Introducing GWT

- Chapter 1 - Introducing GWT
- Chapter 2 - Exercising the GWT Tools: A Simple Application
- Chapter 3 - Creating the Dashboard

PART II: GWT Basics

- Chapter 4 - Interacting with Widgets and Developing New Widgets
- Chapter 5 - Interacting with Panels and Developing New Panels
- Chapter 6 - Interacting with an Application Through Events
- Chapter 7 - Creating Composite Widgets

PART III: Putting It All Together

- Chapter 8 - Creating a Real World Application
- Chapter 9 - Building Client-Side Dashboard Components
- Chapter 10 - Interacting With JavaScript and Existing JavaScript Libraries
- Chapter 11 - Communicating with GWT-RPC
- Chapter 12 - Examining Client-Side RPC Architecture
- Chapter 13 - Alternative RPC Tools: HTTPRequest, RequestBuilder, and FormPanel
- Chapter 14 - Achieving Interoperability in GWT with JavaScript Object Notation (JSON)
- Chapter 15 - Advanced GWT Techniques

PART IV: The Real World

- Chapter 16 - Making Your GWT Application Flexible and Supportable
- Chapter 17 - Testing, Debugging, and Deploying GWT Applications
- Chapter 18 - Migrating and Augmenting Existing Applications
- Chapter 19 - Using Design Patterns Within GWT Development
- Chapter 20 - Using Third Party Information Within GWT Development

- Appendix A – Introduction to Web technologies
- Appendix B – A Quick Java Primer (tentative)
- Appendix C – Overview of Other IDEs and How to Use GWT

1 Introducing GWT

In May of 2006 Google released the Google Web Toolkit (GWT), allowing you to create Ajax applications in Java. This simple change of altering the programming language with which you write applications has far reaching affects. It means that you can build your applications using a real object-oriented language and take advantage of the gazillion Java tools that are already available. Instead of trying to bring tool support to Ajax, Google brought Ajax to a place where the tools already existed.

The need for such a toolkit is rooted in the ever-increasing size of Ajax applications. When you are working on any large application, in any language, as the application becomes larger it also becomes more difficult to maintain. Some languages, like Java, were designed to help solve this problem. Java requires a strictness that is often seen as a hindrance in small applications, but in large applications it excels. While preserving all of Java's benefits, GWT isn't an island, as it doesn't forbid the use of JavaScript. In fact, GWT makes every attempt to be open, allowing it to integrate with existing JavaScript code.

The core feature of GWT is a compiler that can translate your Java code to JavaScript capable of running in a web browser without sacrificing the ability to make "low-level" JavaScript calls. This openness of the toolkit allows you to integrate GWT with popular JavaScript libraries that you might already be using, like Scriptaculous, JSCalendar, and TinyMCE.

GWT's open nature doesn't end there; it also applies to how the application interacts with the server. GWT supports four major techniques for interacting with the server, including JSON, XMLHttpRequest, HTML forms, and its own special flavor of RPC. These four combined allow you access to existing server-side services, written in any language, and makes possible integration with frameworks such as JSF, Spring, Struts, and EJB. This flexibility means that GWT doesn't make more work for you, and it lets you continue to use the same server-side tools that you are using today.

But, still, being able to write Ajax applications in Java isn't enough to make them easier to write. To this end GWT provides support for JUnit, and a special hosted-mode browser that lets you develop and debug in Java without ever needing to deploy your code to a server. This is a real time-saver!

As you can see, GWT is a rich topic, and there is a lot of ground to cover to see it all. In this chapter we will begin the journey through GWT's vast expanse by spending some time enumerating each of GWT's major features. This will take us from compiling, through integration, and ending with testing. Along the way we will provide some very short code snippets to help define some of the features.

After exploring the main features of GWT, we will follow a winding path as we compare GWT to other toolkits. Our hope is to better explain what GWT is, and what it is not, by comparing it to other toolkits. Again, we provide some code snippets for the purpose of

comparison, but not quite a working application – at least not yet, but be patient, it's coming.

At the end of chapter 1, we provide you with a complete example application based on a well known past time. This example will provide you with your first working application and start you in the process of developing with GWT. Taken as a whole, this chapter provides you with a good starting point as you progress through the book and focus on the features that are most important to your specific development needs.

So let's get down to business and find out what GWT is all about, starting with an overview of the primary features that make this toolkit so useful to web application developers.

A Walk Through GWT

GWT provides a rich set of tools focused on solving the problem of moving the desktop application into the browser, but a rich widget set only solves part of this problem. GWT provides the widgets, plus an XML parser, several tools for communicating with the server, internationalization and configuration tools, and a browser history management system. All of this is provided by GWT, along with a Java to JavaScript compiler that will condense your code to have the smallest possible bandwidth footprint.

We begin the exploration with the compiler, the most important piece of the puzzle, along with the accompanying Java emulation library. We then move on to provide information about the rich widget library, and show you how GWT allows you to interface your new GWT code with your existing JavaScript libraries. We then hop over and examine GWT's support for internationalization, and then see what RPC services GWT has to offer. Finally, we wrap up the examination by looking at the XML parser API, browser history management API, and close with a strong dose of JUnit integration. By the end of this section, you should have a good idea of what GWT is capable of, and hopefully will be as excited as we are about this new technology. So let's begin our tour.

Explaining GWT's Java to JavaScript Compiler

The most obvious place to start looking at what GWT provides is the one tool that really defines it, and that is the compiler itself. Compiling your project is done by running the Java program `com.google.gwt.dev.GWTCompiler`, passing it the location of your module definition file along with some other parameters. A module is a set of related Java class and files accompanied by a single configuration file. The module definition will typically include an entry point, which is a class that executed when the application starts.

The compiler starts with the entry point class, following dependencies required to compile the Java code. The GWT compiler works differently than the standard Java compiler, because it doesn't compile everything in the module, instead it only includes what is being used. This is useful in that it allows you to develop a large library of supporting components and tools, and the compiler will only include those classes and methods that are actually used by the entry point class.

The compiler has several style modes which determine what the resulting JavaScript looks like. The default style is "obfuscate", which makes the JavaScript look like alphabet soup. Everything is compressed and impossible to decipher. This isn't being done to prevent it from being read,

although that could be seen as a benefit, but to help keep the resulting JavaScript file small. This is a real concern as your application gets larger and larger. The next style is “pretty”, which generates readable JavaScript. The last style is “detailed” which produces JavaScript code that looks similar pretty code, but includes the full class name in method names to make it easy to trace the JavaScript back to the originating Java code. The pretty and detailed styles are typically used only during development, and obfuscated to reduce the file size used for production.

An important aspect of the compiler is that it compiles from Java source code, not compiled Java binaries. This means that the source for all of the Java classes you are using must be available. This plays a role when you want to distribute GWT code for reuse. When you build distributable jar files you must include both the Java source and compiled Java class files. The GWT compiler also requires that the source code be compliant with the Java 1.4 syntax. This is expected to change eventually, but for now you can't use generics, enums, and other Java 1.5 features in your application. Note that this restriction only applies to code that will be compiled to JavaScript, it does not limit what Java version you can use to write server components that will be communicating with the browser.

One last feature to note is that when your code is compiled to JavaScript it will result in a single JavaScript file for each browser type and target locale. Typically this means that your application will be compiled to a minimum of four separate JavaScript files. Each of these files is meant to run on a specific browser type, version, and locale. A bootstrap script, initially loaded by the browser, will automatically pull the correct file when the application is loaded. The benefit of this is that the code loaded by the browser won't contain code that it can't use. Typically this won't result in a huge bandwidth savings, but in some cases, especially when providing the interface in multiple locale settings, the size can be reduced significantly.

Next we will look at the java emulation library and discover a little more about the limitations of what Java classes the compiler can convert.

Accessing the JRE Emulation Library

When we looked at the compiler we mentioned that to be able to reference a class in our client-side code that we would need the Java source code for that class. This requirement doesn't stop with the use of just external libraries, but also includes the Java Runtime Environment (JRE) as well. To allow developers access to some of the JRE classes, GWT provides the JRE Emulation Library. This library contains the most commonly used parts of the full JRE, which can be used and compiled to JavaScript.

Tables 1.1 and 1.2 list the available part of the JRE, which only includes a select list from the java.lang and java.util packages. If you look through the list carefully you will likely see several missing classes that you consider important. For example, you may notice that the java.util.Date class is available, but not java.util.Calendar, or any date formatting tools.

Table 1.1 Classes from java.lang that are available in GWT

Classes		
Boolean	Byte	Character
Class	Double	Float
Integer	Long	Math
Number	Object	Short
String	StringBuffer	System

Throwable		
Exceptions / Errors		
AssertionError	ArrayStoreException	ClassCastException
Exception	Error	IllegalArgumentException
IllegalStateException	IndexOutOfBoundsException	NegativeArraySizeException
NullPointerException	NumberFormatException	RuntimeException
StringIndexOutOfBoundsException	UnsupportedOperationException	
Interfaces		
CharSequence	Cloneable	Comparable

Table 1.2 Classes from java.util that are available in GWT

Classes		
AbstractCollection	AbstractList	AbstractMap
AbstractSet	ArrayList	Arrays
Collections	Date	HashMap
HashSet	Stack	Vector
Exceptions / Errors		
EmptyStackException	NoSuchElementException	TooManyListenersException
Interfaces		
Collection	Comparator	EventListener
Iterator	List	Map
RandomAccess	Set	

When you start using these classes, you will notice some additional differences. Some of the functionality of these classes differs from the JRE versions in subtle ways. As of this writing, the following restrictions apply:

- Double, Float should not be used as HashMap keys for performance reasons.

- For String.replaceAll, String.replaceFirst and String.split the regular expressions vary from the standard Java implementation.

- StringBuffer(int) behaves the same as StringBuffer().

- System.out and System.err are available, but have no functionality in web mode.

- Stack Traces in Throwable are currently not supported

- The implementation of the Vector class does not include any of the capacity and growth management functionality of the normal Java implementation, nor is there any checking of index validity.

In general, this isn't as limiting as we have made it sound. In many cases you can get around the problem by using other Java classes, writing your own code to perform a specific function, or making direct use of the JavaScript API. As GWT gains momentum, it is likely a lot of these holes will be filled by either the GWT library itself or by open source libraries.

Besides the emulation library GWT provides it's own set of APIs for common tasks related to building rich Internet applications. These APIs, as we will see as we continue through this section, include an XML parser, RPC and international tools, and of course a library of visual user interface components.

Understanding GWT's Widget and Panel Library

GWT ships with a large set of widgets and panels available for use. These fall into three general categories; panels used for layout, widgets that represent HTML equivalents, and rich user interface panels.

With Java's Swing library, layout is handled by attaching a layout manager to a panel. GWT takes a different approach, providing several panels with each display their children in a specific manner. For example, the HorizontalPanel will display its child widgets from left to right, and FlowPanel which will display its children using normal HTML flow rules.

The set of widgets that represent their HTML equivalents includes form type fields like Button, TextBox, TextArea, Checkbox, RadioButton, and FormPanel. In addition to these there are several variations of the HTML table element. These include the base class HTMLTable, and two specialized sub-classes Grid and FlexTable.

GWT also comes with several rich components which are familiar in desktop applications, but not so much in web applications. The TabPanel allows you to place different widgets on different tabs, and the widgets displayed will depend on the currently selected tab, like Firefox's and Internet Explorer's tabbed browsing. The MenuBar provides an easy way to create a multi level menu for your application. Then there is a PopupPanel, DialogBox, StackPanel, Tree, and others.

Although there are over 30 widgets and panels included with GWT, it is likely that you will need a widget that is not available. Fortunately there are many open source widgets already appearing for GWT. This includes calendars, sortable tables, calculators, drawing panels, tooltip panels, and others. There are also a number of widgets available that wrap existing JavaScript libraries, like the Google Maps API, Google Search API, and Scriptaculous effects. Besides HTML-based widgets there are also widgets available for Scalar vector Graphics (SVG). We will list some of these open source projects later in the book in chapter 20.

When building your own widgets by extending those that come with GWT it is often required that you access the browsers underlying JavaScript objects. It would be great if we could forget about the underlying JavaScript completely, but that isn't always the case. Next we will look at the facilities GWT provides for interfacing Java with the underlying JavaScript.

Using JSNI to Execute JavaScript from Java

Because GWT represents an abstraction on top of JavaScript, there will be times when you need to write code at the JavaScript level. There are several reasons why you might want to do this, including the need to use existing JavaScript code, or to use a browser API that isn't accessible through existing GWT utilities.

JSNI allows you to execute JavaScript from Java, as well as execute Java from JavaScript. You can also pass Java objects to JavaScript methods. We will get into the finer points of doing this later in the book, but here are some examples to give you an idea as to how this works.

```
public native int addTwoNumbers (int x, int y)
/*-{
    var result = x + y;
    return result;
}-*/;
```

This first example is very basic but reveals how the mechanism works. In Java you can declare a method as "native", meaning that the actual implementation of the method will be written in some other language. When you declare a method as being native you are not allowed to specify a code block for the method, the same as a method declaration in a Java interface. When you inspect this method you will see that what appears to be a block of code is really all contained in a

multi-line comment. Inside of this comment is the “native” JavaScript code that will be executed when the method is called. This satisfies the Java syntax requirement of not allowing a code block for native methods, yet provides the actual JavaScript that can be used by the GWT compiler to allow execution of this code.

```
public native void fillData (List data)
/*-{
    data.@java.util.List::add(Ljava/lang/Object;) ('item1');
    data.@java.util.List::add(Ljava/lang/Object;) ('item2');
}-*/;
```

In this example we are passing a Java List object to the method, and using JavaScript to add two items to it. Because we are calling the add() method on a Java object we need to use a special syntax to provide details on the object and method we are referencing. Here we let the GWT compiler know that the variable data is an instance of java.util.List, and that the add() method takes a single java.lang.Object argument. This mechanism is fairly easy to use and allows you to include any JavaScript code in the same source file as your Java code.

We will get into the finer points of using JSNI later in the book. Now we want to switch gears a bit and look at how GWT handles internationalization and how those same tools can be used for application configuration.

Examining GWT's Internationalization and Configuration Tools

Several techniques are provided by GWT that can aid with internationalization and configuration issues. This may seem like an odd pair, but they are similar in that you want the ability to store text strings or numeric values in a properties file, and access them from your application. GWT provides two primary mechanisms that should handle most needs: static inclusion at compile time, and dynamic inclusion at run time.

When including your setting statically, you do so by implementing an interface, either Constants or Messages, and create a single method for each property that you will want to use. For example, perhaps we would want to use a properties file to store our welcome message along with the image path of the logo for our application, in which case we would provide a method for each in our interface.

```
public interface MySettings extends Constants
{
    String welcomeMessage();
    String logoImage();
}
```

Once we have our interface GWT can be used to dynamically create an instance of this interface, and it will automatically attach the properties file settings to it. We can also set up several properties to be used for different locales, for example if we wanted the text to change based on the language of the reader.

The Messages interface differs from Constants in that you may specify arguments to the methods which are then used to fill placeholders in the property text. For example, we might want to alter the interface above to allow the person's name to be included in the greeting message. Our properties file might look like the following.

```
welcomeMessage = Welcome to my book {0} (1)
logoImage = /images/logo.jpg
```

The placeholder {0} is used to mark the place where the first variable should be inserted into the message, and {1} for the second. The interface that we used for Constants would need to be modified to use the Messages interface instead, and we would add two arguments to our method.

```
public interface MySettings extends Messages
{
    String welcomeMessage(String fname, String lname);
    String logoImage();
}
```

One of the benefits of using these two interfaces is that the messages from your properties file are statically included in your compiled JavaScript. This means that the performance of using a value from a properties file is about the same as using a hard-coded string in the application. The compiler will also only include properties that are actually referenced. This makes it possible to use only a few properties from a very large properties file without having all of the properties embedded in your JavaScript code.

If you are only using this mechanism for settings then you will likely only have a single properties file, but if you want to provide for localization you will have one properties file for each supported language. When compiling your code with multiple properties files the GWT compiler will create a different set of JavaScript files for each locale supplied. If you are supporting a lot of locales this will result in a lot of files, but the benefit is that each JavaScript file will only have a single set of properties embedded in it making the file size smaller.

The second mechanism provided by GWT is the Dictionary class, which doesn't use properties files at all. Instead the Dictionary object can be used to grab settings that have been embedded in the HTML as JavaScript objects.

```
var MySettings = {
    welcomeMessage: "Welcome to my book",
    logoImage: "/images/logo.jpg"
};
```

In the GWT application you would use the Dictionary class to load by specifying the JavaScript variable name. This is done by calling the `getDictionary()` method which returns an instance of the Dictionary class. You can then use the methods of the object to get the individual settings. This mechanism is idea when you want to pass data to your GWT application.

```
Dictionary settings = Dictionary.getDictionary("MySettings");
String logo = settings.get("logoImage");
```

The configuration mechanisms provided by GWT will handle most of your configuration and internationalization issues, and allows for both dynamic and static properties. Next we will take a look at one of the most important features of a rich Internet application, that being remote procedure calls.

Calling Remote Procedures with GWT

Most non-trivial GWT applications will need the ability to communicate information between the browser client and the server. In this section we will highlight some of the RPC features included with GWT. Earlier in this chapter we discussed the XMLHttpRequest object that is available in most modern browsers, and this object is the basis of all browser based RPC. This object has the ability to send a request to the web server, and receive its response. This is the only mechanism available to web developers without using proprietary browser technologies or plug-ins. Unfortunately all this mechanism can do is pass data to the web server, the same way an HTML form can, and receive a web page as a response. This is unfortunate because it doesn't allow us to send programming objects between the client and server. As we will see below, GWT uses serialization to circumvent this problem.

In the GWT example below we are exercising the XMLHttpRequest object found in GWT, which is just a wrapper around the browser's underlying XMLHttpRequest object. You will see that we are calling a JSP page and processing the resulting string data. String data is the only type of data that can be returned using this object.

```
XMLHttpRequest.asyncGet("getdata.jsp", new ResponseTextHandler()
{
    public void onComplete (String responseText)
    {
        // process here
    }
});
```

Before we go on to explain the two frameworks that allow you to pass Java objects, we need to discuss the asynchronous nature of the XMLHttpRequest object. By asynchronous we mean that the application will not be blocked after a request is sent to the server, meaning processing will still continue. Typically when we call a method in a program we expect the blocking behavior, meaning processing will not continue until the execution of the method completes, and possibly returns a value. If this type of execution is new to you it might take a little bit to get used to. This behavior is beneficial though because your program will not be held up waiting for the request to complete, and it is still possible to make the request blocking. We will discuss techniques on how to do this in the RPC chapter.

The primary mechanism that GWT provides on top of the XMLHttpRequest object is a proprietary mechanism only found in GWT. It allows you to pass any object to and from the server that implements the GWT IsSerializable interface. There is some coding required to setup the connection, similar to what you would need to create a database connection with JDBC. Once the boilerplate code is written you can call the server as simply as you would call another method in the application.

In this example below we left out a boilerplate code, which consists of only a few lines, just so that we can concentrate on the actual syntax of the call. The `service` variable is our connection to a specific class on the server, and when we call `changePassword()` the request is serialized and sent off to the server for processing. Here we are passing two arguments plus a response handler. These arguments can be of any serializable type, and there can be any number of arguments.

```

service.changePassword("abc123", "m@tr1x",
    new AsyncCallback()
    {
        public void onSuccess (Object result) {
            Window.alert("password changed");
        }

        public void onFailure (Throwable ex) {
            Window.alert("uh oh!");
        }
    });

```

When a successful result is returned, which is done asynchronously, our `onSuccess` method handler is executed. Notice that the `onSuccess` method receives a generic `Object`, and not a more specific object type. This allows the server to return any serializable object back to the browser, and not just string data. The types of objects that can be passed include the usual suspects, like `Map` and `List`, arrays, user defined classes, and more.

The second mechanism provided by GWT is a set of classes that can be used to represent and parse objects encoded with the JavaScript Object Notation (JSON). JSON is an encoding scheme that was designed specifically to make it easy to be produced and consumed by JavaScript code in the browser, which means that it performance friendly.

JSON is different from the proprietary RPC mechanism is because you can only send and receive a small set of defined JSON objects. The content types include array, map, number, string, and Boolean. This is a pretty small set of objects but it has one important advantage over the proprietary mechanism, and that is that there are server-side implementations written in dozens of languages. This allows you to communicate directly with services written in Python, Perl, PHP, Ruby, and other non-Java languages. So if you prefer to write your server-side code in something other than Java, the JSON API allows you to do this rather painlessly.

```

HTTPRequest.asyncGet("data/test.cgi", new ResponseTextHandler()
{
    public void onCompletion (String responseText)
    {
        try {
            JSONValue value = JSONParser.parse(responseText);
            JSONString strVal = value.isString();
            if (strVal != null) {
                Window.alert(strVal.stringValue());
            }
        }
        catch (JSONException e) {
            Window.alert("uh oh, parse error!");
        }
    }
});

```

In the code example we are calling a CGI script and parsing the result with the `JSONParser`. The result can be any of the JSON value types, and JSON provides methods allowing you to inspect the value to determine the specific type. Here we have called `isString()` on the value object which will either cast the object to a `JSONString`, or return

null if the object isn't a string value.

Sending a JSON request is done in reverse, where you create JSON objects, then call the `toString()` method to serialize the object. You would then pass the serialized result to the server-side application using the `HttpRequest` object.

These three mechanisms combined allow you to communicate pretty much anything you might need to between the client and server. There is one small issue that we have avoided until now, and that is that both the GWT compiled code and the RPC service on the server must originate from the same domain. This is a security feature of modern browsers which prevent what they call cross-domain attacks. This has presented an issue for some developers who wish to consume remote services that don't originate from their server. It is possible to perform this sort of communication without circumventing browser security by using a proxy. We will provide an example of using a servlet to proxy data in chapter 14 when we look at JSON.

Next we will take a look at GWT's XML parser, which is often used in conjunction with RPC, allowing you to receive data from the server as XML and using the GWT XML parser to read that data.

Investigating GWT's XML Parser

GWT provides an XML parser that returns a Document Object Model (DOM) representation of the data. If you haven't worked with the XML DOM that comes with later versions of Java, it essentially creates a tree from the XML document. You use the DOM API to traverse the tree and potentially modify it.

GWT takes advantage of the fact that modern browsers have the ability to parse XML and can create the DOM. Because the parsing is done by the browser and not GWT, you get the performance benefit of native code execution. The flip side of the coin is that there is no stream-based parser like SAX. Stream-based parsers are useful when the XML document is extremely large, allowing you to process the document in parts. DOM on the other hand requires a complete parse of the document.

```
HttpRequest.asyncGet("data/test.xml", new ResponseTextHandler()
{
    public void onCompletion (String responseText)
    {
        Document doc = XMLParser.parse(responseText);
        Element root = doc.getDocumentElement();

        NodeList children = root.getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            processNode(children.item(i));
        }
    }
});
```

In this example we are combining the functionality of the `HttpRequest` object, which we touched on in the previous section, and the XML parser. We are requesting a file from the server called "data/text.xml", parsing it, and processing it. In this case we are using a static file which may contain application settings, but we could also point this to a dynamic JSP or CGI which will build an XML response on the fly. This is the mechanism that is being used to allow GWT to

talk with pre-existing server applications which use XML-RPC for communication.

Next we will go back to something that the user can see. That something is the browser history that can be navigated by the user via the back and next buttons in the browser.

Managing the Browser History

One of the early complaints of Ajax is that it “broke” the Back button on the browser. What that means is that when an Ajax application updates the contents of the page it does so without involving the browser interface. So, even though the contents of the page have been changed by the Ajax application, the browser still sees it as being the same page. This has a tendency to confuse some users.

There is a solution to this problem, but it tends to be a little complicated to use. It requires adding a hidden frame to your page, and some amount of scripting to get it to work. What GWT has done is to build this system for you and make it available via a simple event interface. To access it you need to write an event handler that implements the `HistoryListener` interface and add “token” names to links in your page. When a link is clicked, the `onHistoryChanged()` method is called, the browser is notified of the content change, and the token name of the link is passed to the handler.

```
public void onHistoryChanged (String historyToken)
{
    if (historyToken.equals("overview")) {
        // display overview panel
    }
    else if (historyToken.equals("reports")) {
        // display reports panel
    }
    else if (historyToken.equals("documents")) {
        // display documents panel
    }
}
```

Clicking the browser’s back button causes the `onHistoryChanged()` method to be called with the appropriate token. We have been concentrating on the Back button, but the Forward button will also work as you would expect. Using the `History` class, you can also programmatically affect the browser’s history. You can programmatically navigate both forward and backward and add a new token to the history. The details on implementing this, and some things to look out for, are covered in chapter X.

Introducing GWT’s JUnit Integration

It has always a best practice to test your code, and various frameworks have been made available over the years to make the process less painful. JUnit is one of the more popular tools used by Java developers, and is integrated with many IDEs. Instead of creating a new framework for GWT, Google has provided JUnit support for testing your GWT applications, allowing you to take advantage of the existing JUnit integration.

To create a new test case, you create a class that extends `GWTTestCase`, which in turn JUnit’s `TestCase` class. You then implement the one required method, `getModuleName()`, followed

by any number of specific tests. The module name is used by GWT, telling it where it can find the configuration file for the project. Other than this one new method, it is a standard JUnit test case.

```
public class MathTest extends GWTTestCase
{
    public String getModuleName ()
    {
        return "org.mycompany.MyApplication";
    }

    public void testAbsoluteValue ()
    {
        int absVal = Math.abs(-5);
        assertEquals(5, absVal);
    }
}
```

When you test your code, there is a little more going on behind the scenes than we have led on. In chapter X we will explain how your code is really tested and show you how you can chain tests together into a test suite.

Up to now we have only looked at how GWT solves the hardships associated with rich Internet applications. GWT isn't the only game in town, though, and we are often asked how GWT fares against other technologies. In the next section, we look at some technologies GWT is usually compared with.

GWT vs. Other Solutions

GWT isn't the first tool trying to make it easy to build rich Internet applications, and it undoubtedly won't be the last. In this section we take a look at several other technologies that are often lumped into the same category as GWT, and explain where the differences lie.

In each of the following sections we provide a code example for creating a text box and a button. We then fill the text box with the word "clicked" when the button is clicked with the mouse. Finally, we compare the code example provided to our GWT reference implementation and discuss the important differences. Figure 1.1 shows what the GWT reference implementation looks like in a web browser when the application starts, and after the action button is clicked.



Figure 1.1: Images of the GWT reference implementation of text box and button before and after the action button is clicked

To begin, we need to examine our GWT reference implementation.

```
final TextBox text = new TextBox();           | #1
text.setText("text box");                     | #1
```

```

final Button button = new Button();           | #2
button.setText("Click Me");                 | #2

button.addClickListener(new ClickListener()  | #3
{
    public void onClick (Widget sender)     | #3
    {
        text.setText("clicked");           | #3
    }
});                                         | #3

Panel main = new FlowPanel();               | #4
RootPanel.get().add(main);                 | #4

main.add(text);                             | #5
main.add(button);                           | #5

```

- #1 Creates text box>**
- #2 Creates button>**
- #3 Attaches event handler>**
- #4 Attaches main panel >**
- #5 Attaches widgets to panel >**

This is the first example of GWT code that we have looked at, and it deserves a thorough explanation. #1 To start, we create a new text box, which once displayed in the browser will look like a normal text box that you might find in a form on the web. #2 Next we create a button, and set the text to display on the button. Note that at this point neither the text box nor button has been “attached” to the web page. #3 We then add an event handler for the button. We do this by adding an click listener object which will be called when a user clicks on the button. You will notice that the object we add as a listener is an anonymous class. If you haven’t seen this type of Java construct before it might seem a little unnatural at first, but all we are doing is creating a new object that implements the interface ClickListener. #4 Next we add a panel to our window, and then #5 add the text box and button to our panel.

When we run this GWT code, it renders an HTML page with a text box and button, and when the button is clicked changes the text in the text box. In the following sections, we will port this very simple code to a few other frameworks and discuss how they differ from GWT. Note that we are making every attempt to be unbiased as we describe the competing frameworks. We believe that each tool is different from the next, and each has its own strengths and weaknesses.

With that in mind we will begin our tour with Swing, followed by Echo2, Java Server Faces, and Ruby on Rails.

GWT Vs. Swing

Swing is the standard toolkit for building GUI applications in Java. This might seem like a strange choice of tools to compare GWT to because Swing isn’t typically associated with web applications. The reason for the comparison to GWT is because the two frameworks are very similar in how you write code for them.

```

final JTextField text = new JTextField();           | #1
text.setText("text box");                          | #1

final JButton button = new JButton();              | #2
button.setText("Click Me");                        | #2

button.addActionListener(new ActionListener()      | #3
{
    public void actionPerformed (ActionEvent e)    | #3
    {
        text.setText("clicked");                  | #3
    }
});                                                | #3

final JFrame rootPanel = new JFrame();
Panel main = new Panel();                          | #4
rootPanel.getContentPane().add(main);              | #4

main.add(text);                                    | #5
main.add(button);                                  | #5

rootPanel.setVisible(true);
rootPanel.pack();

```

- #1 Creates text box>**
- #2 Creates button>**
- #3 Attaches event handler>**
- #4 Attaches main panel >**
- #5 Attaches widgets to panel >**

This Swing code should look vaguely familiar to the GWT reference example, in fact it should look nearly identical. There are a few name changes, for instance GWT's ClickListener interface is called ActionListener in Swing.

For Swing developers there are a few important differences between GWT and Swing. The first is that the components that ship with GWT don't follow the Model View Controller (MVC) pattern. That is to say that there isn't a model object that can be shared by multiple components to keep them in sync. The second difference is that GWT doesn't use layout managers for controlling the layout. Instead you will use panels that have built in layout styles. For example, the GWT HorizontalPanel will arrange it's child components left-to-right across the page, while the DockPanel allows you to add widgets to the panel in a similar fashion to the Swing's BorderLayout.

That being said, these differences are fairly easy to work with, and GWT is a very friendly environment for Swing developers. Next we will look at Echo2, which allows you to write applications in a similar manner to GWT, but takes a very different approach.

GWT Vs. Echo2

Echo2 is similar to GWT in how it is used to create the user interface. You use the API create instances of components, then add them to the display. Below is the Echo2 version of the GWT reference example. You will see that the Echo2 version looks nearly identical to the GWT version.

```

final TextField text = new TextField();           | #1
text.setText("text box");                       | #1

final Button button = new Button();             | #2
button.setText("Click Me");                    | #2

button.addActionListener(new ActionListener()  | #3
{
    public void actionPerformed (ActionEvent evt) | #3
    {
        text.setText("clicked");                | #3
    }
});                                             | #3

Window window = new Window();
window.setContent(new ContentPane());
Row main = new Row();                           | #4
window.getContent().add(main);                 | #4

main.add(text);                                 | #5
main.add(button);                              | #5

```

- #1 Creates text box>**
- #2 Creates button>**
- #3 Attaches event handler>**
- #4 Attaches main panel >**
- #5 Attaches widgets to panel >**

Although both frameworks use similar APIs, they work in an entirely different fashion. Applications written for Echo2 run on the server, not the client. With GWT you compile your Java source to JavaScript, and run it on the browser. With Echo2 you compile your Java source to Java class files, and run them on the server. This also means that when a client-side event is triggered may need to be handled on the server.

The consequence of this is that an interface built with Echo2 will need to hit the server more often, but it doesn't need to deal with an RPC API since the RPC just happens all by itself. It also means that Echo2 doesn't need to send all of the JavaScript to the browser at once, it only sends what it needs to given the current state of the application. And, finally, this also means that you are tied to using a Java application server since this is required to host your Echo2 application.

Next up is another Java based framework called Java Server Faces.

GWT Vs. Java Server Faces

Java Server Faces (JSF) is a web framework for Java-based web applications. It makes use of managed Java beans on the server, which represent the model, plus a set of tag libraries which can be placed in an JSP page to reference the properties of the model. In a standard JSF implementation, all of the processing is done on the server, and the web page reloads for each transaction. The fact that the page needs to reload for each transaction doesn't really make JSF a viable rich client for built-in components, but with additional effort it is possible. For the sake of comparison we will provide a standard non-Ajax JSF application that can perform the same action as our reference GWT application.

The first step in creating a JSF application is to create a class to represent the model. For our example, our model is very simple; it contains only one property named "text". In standard Java

bean fashion, we make the property private to the class and provide accessors for getting and setting the value. To this we also need to add a method named `changeText()`, which will be triggered when the command button is clicked.

```
package org.gwtbook;

public class SampleBean
{
    private String text = "text box";

    public String getText ()
    {
        return text;
    }

    public void setText (String text)
    {
        this.text = text;
    }

    public void changeText ()
    {
        this.text = "clicked";
    }
}
```

The next step is to register this class as a managed bean in the JSF configuration file. We provide the name “sampleBean” for our managed bean which will be used to reference it in the JSP code to follow.

```
<managed-bean>
  <managed-bean-name>sampleBean</managed-bean-name>
  <managed-bean-class>org.gwtbook.SampleBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

The JSP page will look similar to a standard JSP page. It uses two JSF tag libraries to specify the view and controls that we are using. For the value in the `inputText` tag we reference the `text` property of our managed bean using the JSF expression language. In the `actionButton` tag we see it again, but this time it references our `changeText()` method.

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<f:view>
<html>
<head>
  <title>JSF Example</title>
</head>
<body>

  <h:form>
    <h:inputText value="#{sampleBean.text}"/>
    <h:commandButton
```

```
        value="Click Me"  
        action="#{sampleBean.changeText}" />  
    </h:form>  
  
</body>  
</html>  
</f:view>
```

JSF is very different from GWT in the fact that JSF provides very little in the way of support for rich client-side functionality. It would be possible to build reusable client-side components, doing some of the work in JavaScript, but custom components would have little reuse value. Because JSF does integrate the client-side, it is in competition with GWT, but there is still some potential for integrating the two.

Next we will compare GWT to Ruby on Rails, a very popular non-Java framework for writing web applications.

GWT Vs. Ruby on Rails

The title of this section is a little misleading because GWT and Ruby on Rails don't really compete, although they may overlap in some respects. Ruby on Rails is a rapid development framework using the Ruby language. It provides the server-side of the equation and is specifically designed to handle a lot of the back-end work automatically for you. On the client-side, Ruby on Rails provides some support for Ajax, allowing you to use the Java equivalent of a tag library in your HTML code. The end result is that Ruby on Rails can send data to the server triggered by a user action, and display a response in the page. It is not designed, though, for complex interactions between the client and server.

GWT is client-centric, and most of what GWT does is on the client-side of the picture. It allows you to develop and display widgets using Java, and write Java handlers to trap user triggered actions. GWT can communicate with the server, as needed, which could be driven by user interaction, or perhaps a timed event. GWT then allows you to compile all of the Java code to JavaScript so that the program can be run in the browser. On the server, GWT only provides a mechanism for serializing and deserializing Java objects so that they can be received from the browser and sent back, and does not get involved in other aspects of the server.

Instead of competition between GWT and Ruby on Rails, we find an opportunity for integration. This is in part driven by the fact that GWT provides several non-proprietary schemes for passing data between client and server. We are finding that many developers who are starting to use GWT are using non-Java technologies on the server, and are looking at GWT to provide only client-side functionality.

Building Your First GWT Application

As most developers, the first thing we want to do when trying out a new technology is to build something with it. We could write pages and pages explaining how to use GWT, and we will, but that can't replace having an actual running example in front of you. To that end, this section will be drastically light on the details, and we will focus only on getting your first GWT application up and running.

In this section we assume that you have downloaded and uncompressed the appropriate GWT distribution for your platform, and verified that you have a version of Java installed on your workstation that can be used with GWT. If you haven't, you can download GWT from <http://code.google.com/webtoolkit/> and Java from <http://java.sun.com>.

So let's get to it and build something.

Building and Running an Example Application

GWT ships with an application creator tool that creates a directory structure and populates it with some sample code. This tool is your friend; you will likely use it to create the skeleton for every GWT application you write. We will cover this in much greater detail in the next chapter.

To run it, you will need to open a command prompt or shell and navigate to the directory where you unpacked GWT. As arguments to the command, we specify an output directory named "Sample" and specify the Java class that we want the tool to create.

```
1.1 applicationCreator -out Sample org.sample.client.App
```

The output of this command will look like this on Windows, and similar on all other platforms:

```
1.2 Created directory Sample\src
1.3 Created directory Sample\src\org\sample
1.4 Created directory Sample\src\org\sample\client
1.5 Created directory Sample\src\org\sample\public
1.6 Created file Sample\src\org\sample\App.gwt.xml
1.7 Created file Sample\src\org\sample\public\App.html
1.8 Created file Sample\src\org\sample\client\App.java
1.9 Created file Sample\App-shell.cmd
1.10 Created file Sample\App-compile.cmd
```

That's it. You have now written your first fully functional GWT application.

Granted, we haven't actually seen it run yet, but rest assured that this little application will compile to JavaScript and execute. At this point, before you run it, we suggest that you explore each of the generated files. Below we provide a description of each file.

App.gwt.xml	This is the configuration file for the "module". This is used to define the entry point class, dependencies, and compiler directives. The entry point class is the class that gets executed when the module loads into your browser.
App.html	This is the HTML page that will load and execute the application. We don't want to get into the specifics as to how the HTML loads the module, but it may be worth looking at. This file is well commented.

App.java	This is a sample entry point class that is generated by the creator tool. This is where you put the Java code that you want to be compiled to JavaScript.
App-shell.cmd	This is a simple shell script that executes the <i>hosted-browser</i> that ships with GWT. The hosted-browser works like your web browser, but it is specifically tailored for GWT development. Executing this will launch the application.
App-compile.cmd	This is another simple shell script that executes the Java to JavaScript compiler. Just like the App-shell script, this script references the module configuration file which provides the details on what needs to be compiled. Running this will create a directory “www” and generate the JavaScript there.

The next step is to actually run the application, and, if you haven't guessed already by looking at the generated source code, it is a “Hello World” application. You have two options for doing this. You may either (A) run the App-shell script to launch the hosted-browser, or (B) run the App-compile script, then open the App.html file in the “www” directory in your browser. We recommend that you do both. It is worth experimenting here a little to see how each of these works, because you will use both of these frequently when you develop your own applications.

In figure 1.2 we show what the application looks like when run in Firefox. It is a very basic application that presents a button that toggles the visibility of the “Hello World!” message. From browsing through the project files, you may have noticed that the text at the top of the page is in the HTML page, and only the button and “Hello World!” label are referenced in the Java file. This is a good example as to how GWT can mix HTML content with application logic, allowing you to leverage the skills of a designer without needing to teach them GWT.

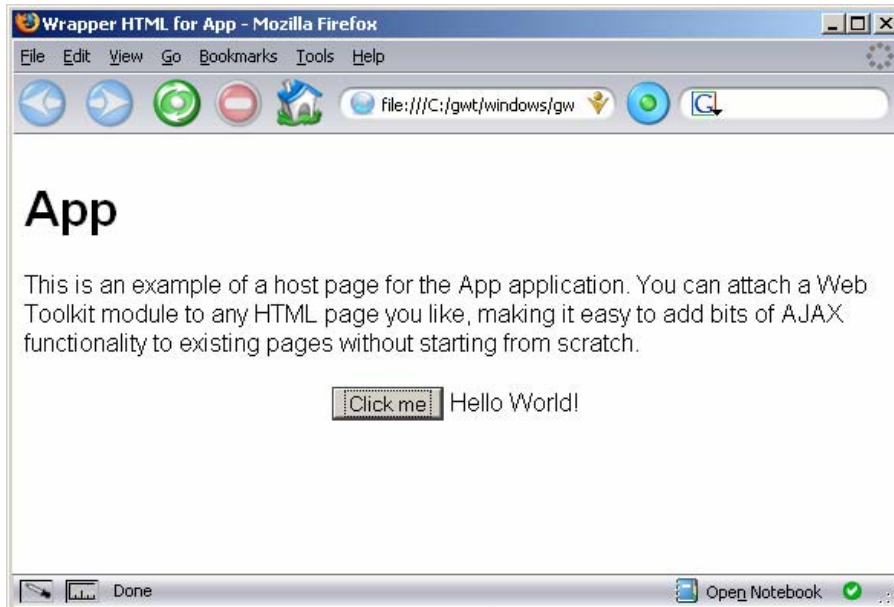


Figure 1.2 The sample generated Hello world application running at JavaScript

That is a wonderful example, but, if you are like us, you have lost count of how many Hello World examples that you have seen. So let's have some fun and alter the code a bit to do something a little more exciting.

Building Tic-Tac-Toe with GWT

First things first; if you have a favorite IDE, now is the time to start it up. You will need to add the gwt-user.jar file that came with the GWT distribution to your classpath. It doesn't matter too much what IDE you use at this point, but it worth noting that GWT does provide some additional support for Eclipse users, as we will see in chapter 2. You should also set the project to use Java 1.4 compatibility. As of writing this, GWT does not yet support the new Java 5 syntax constructs for client-side code, so for now we need to limit ourselves to Java 1.4. The client-side code is considered to be any Java code that will be compiled into JavaScript and run in the browser.

The next thing we need to do is decide what to build. This early in the book we don't want to overwhelm you with advanced GWT concepts like internationalization or remote procedure calls, so we will stick to something simple. So what we will do is build a very simple rendition of Tic-Tac-Toe.

Designing the Tic-Tac-Toe Game

We will build a Tic-Tac-Toe game by starting with the Grid widget from the GWT library. The Grid control will render a HTML table with a specific number of columns and rows. In our example we will use this to build the regulation 3x3 Tic-Tac-Toe grid, but you may alter this to use a more challenging 4x4, or even 5x5 grid.

In each of the cells of the Grid control we will put a blank Button control. The end result looks something like Figure 1.3. Unless you are using a visual designer to build your GWT application, it is usually a good idea to sketch out your application before you build it, like we did

with this diagram.

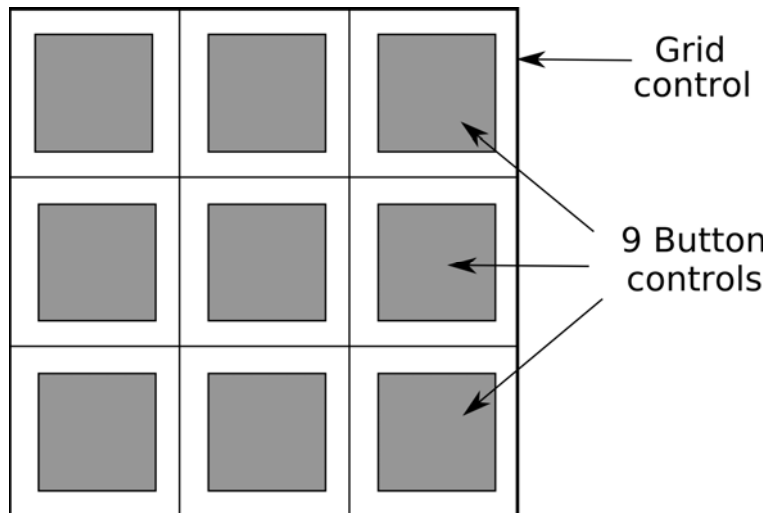


Figure 1.3 A diagram of the GWT controls used in a Tic-Tac-Toe game

The logic of the application is very simple. When the first player clicks a button, we change the text of the button to display an “X”. When the next player clicks a different button, we change the text of the button to display “O”. We toggle back and fourth between “X” and “O” as the game progresses.

We will also need to perform a check before changing the text of a button. If the button that was clicked by the player already contains a marker, then we display an alert message that tells the player that they must select a different square.

All of the changes we will be making are in the App.java file, so if you want to follow along with us you should open that file now. Now let’s move on and write the code to make this work.

Writing the Tic-Tac-Toe Game

To build our Tic-Tac-Toe game, we need to import the needed Java classes for the project. In the App.java class, you need to include the following import lines for the project.

```
1.11 import com.google.gwt.core.client.EntryPoint;  
1.12 import com.google.gwt.user.client.Window;  
1.13 import com.google.gwt.user.client.ui.*;
```

The EntryPoint interface is required to be implemented by the main class of the module. The EntryPoint interface requires the implementation of the method `onModuleLoad()`. This method has the same purpose as the `main()` method in a regular Java application, or the `service()` method in a Java servlet. This is the method that starts the application.

The Window class is roughly equivalent to the window JavaScript object. It provides access to the browser window to do things such as determine the browser’s height and width, or to display an alert message to the user.

The third import we use is to import the entire GWT user interface package. This includes dozens of widgets, listeners, and panels that we will see more of as you progresses through the

chapters. For now we won't get into any of the details of what classes this includes, and we will just focus on the project at hand.

Listing 1.1 Tic-Tac-Toe code

```
1.14 public class App implements EntryPoint {
1.15
1.16     private boolean playerToggle = true;           | #1
1.17
1.18     public void onModuleLoad() {
1.19         Grid grid = new Grid(3, 3);                 | #2
1.20
1.21         for (int col = 0; col < 3; col++) {
1.22             for (int row = 0; row < 3; row++) {
1.23                 Button button = new Button();       | #3
1.24                 button.setPixelSize(30, 30);       | #3
1.25
1.26                 button.addClickListener(new TicTacClick()); | #4
1.27
1.28                 grid.setWidget(col, row, button);  | #5
1.29             }
1.30         }
1.31
1.32         RootPanel.get().add(grid);                 | #6
1.33     }
1.34
1.35     private class TicTacClick implements ClickListener { | #7
1.36         public void onClick(Widget sender) {       | #7
1.37             // todo                                 | #7
1.38         }                                           | #7
1.39     }                                               | #7
```

[#1] We begin by defining a Boolean value to keep track of whose turn it is so that we know if we should display an “X” or “O”. [#2] Next we create a 3x3 Grid control. At this point the control won't be displayed on the page, but it is constructed in memory. [#3] As we loop through the 3x3 Grid, we create the Button control that will appear in that square. We also set the width and height to 30 pixels. Alternatively, we could have used CSS in the HTML page to specify the height and width, but for now this will suffice.

[#4] After we create the Button control, we register a handler to receive click events for the Button. We have defined a private class called TicTacClick [#7], which will handle this event, but for now we haven't provided any code for this.

[#5] Once the Button has been created, we need to add it to the Grid. Here we use the col and row loop variables to place it in the right cell.

[#6] Once the Buttons are added to the Grid, we need to add the Grid to the page. The RootPanel class represents the top level panel on the HTML page, and here we add the Grid to our RootPanel. By calling RootPanel.get() we are adding the Grid to the very bottom of the page. Alternatively you can pass a String argument to the get() method to specify the HTML element ID where the control should be added. This allows you to add controls to not only the end of the page, but inside of any HTML element that has an ID attribute.

[#7] This is our code that will handle the Button clicks and make the “X”s and “O”s appear. We didn’t want to try to put all of the code into a single listing, so we will add the look at this next. You should replace segment [#7] with the following:

```
1.40 private class TicTacClick implements ClickListener {
1.41
1.42     public void onClick(Widget sender) {           | #1
1.43         Button button = (Button) sender;         | #1
1.44
1.45         if (button.getText().equals("")) {       | #2
1.46
1.47             if (playerToggle) {                 | #3
1.48                 button.setText("X");           | #3
1.49             }                                     | #3
1.50         else {                                   | #3
1.51             button.setText("O");               | #3
1.52         }                                       | #3
1.53         playerToggle = !playerToggle;         | #3
1.54     }
1.55     else {
1.56         Window.alert("That square is already taken "); | #4
1.57     }
1.58 }
1.59 }
1.60
```

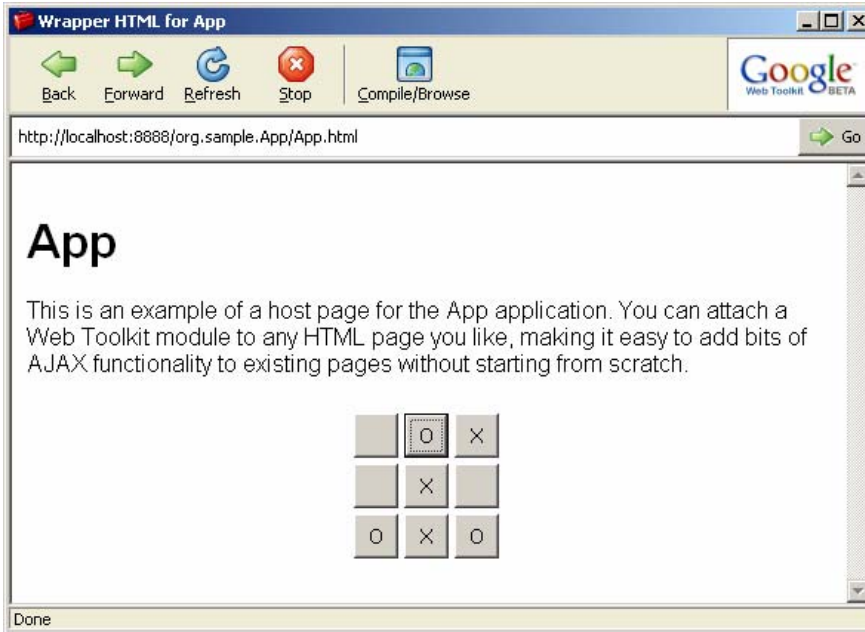
[#1] Our TicTacClick class implements ClickListener, which requires that we implement the method onClick(). This method is called whenever a click event is triggered, and as an parameter to this method we receive a reference to the object that was clicked. In our case we know that it is a Button control that was clicked, so we can cast it as such.

[#2] We need to test if this specific Button has been clicked before, and if it has we need to alert the player that they need to select a different square. The getText() property of the Button will return the text that is displayed in this control.

[#3] Remember the boolean toggle property that we created in our class? We created it to toggle between player turns. If the value is true we display an “X”, and if it is false we display an “O”. We then toggle the playerToggle value so that it switches the displayed value the next time a Button is clicked.

[#4] If the Button is already displaying text we will drop down to this else condition. Here we are using the Window object, which is the equivalent of the JavaScript window object, to display a message to the user indicating that they need to select a different square.

If you typed everything in exactly the same as we did, your Tic-Tac-Toe game should look like figure 1.4, which shows the game in the hosted-mode browser.



1.61

Figure 1.4 A Tic-Tac-Toe GWT application running in the hosted-mode browser.

1.62

What we have done with this example is pretty simple, but we hope that it has given you a small taste of what GWT is capable of. This example only scratches the surface, though, as there is much more to GWT than simple client-side applications.

In any case, we need to wrap things up and start getting into the fine details. So let's see if we can summarize what we have seen thus far and prepare ourselves for the rest of the GWT adventure.

Summary

The Google Web Toolkit (GWT) adds a new tool to the web developer's tool belt, helping to solve some of the hardships involved with developing complex rich Internet applications. GWT changes the way you write rich clients by allowing you to write web applications the same way you write desktop applications with Swing. With GWT, you write code in Java, using a plethora of fancy Java tools, without the need to learn the intricacies of JavaScript. It provides an abstraction on top of the DOM, allowing you to use a single Java API without having to worry about differences in implementations across browsers. When you are done with your Java application, you can then compile your code to JavaScript, suitable for running on today's popular browsers.

Tools like Echo2 attempt to do this as well, but they require a Java application server to serve the application. GWT allows you to instead create an application that compiles completely to JavaScript and can be served by any web server. This allows GWT to be easily integrated with existing applications no matter what type of server you are running.

An important part of what GWT has to offer is its toolset for making Remote Procedure Calls (RPC). It provides a simple RPC mechanism for passing Java objects between the client and server. It does so by serializing and deserializing the objects on both client and server, allowing

you to pass custom beans without having to worry about serialization details. GWT also allows communication to non-Java applications via the standard JavaScript Object Notation (JSON). JSON libraries are available for most languages, making integration a relatively simple task.

GWT as a whole lets you develop web applications at a higher level of abstraction, and leverages the tools already available for the Java language. It provides an easier way to build rich Internet applications. In the chapters that follow, we will get into the details of doing this. We will show you how to set up your development environment, build an application with existing widgets, create some new widgets, communicate with the server, and much more. First, let's start by expanding upon the default application we began in Chapter 1, then applying that to a running Dashboard example we will be returning to throughout the book.