Ajax for Java developers: Exploring the Google Web Toolkit

Develop Ajax applications from a single Java codebase

Level: Advanced

Philip McCarthy (philmccarthy@gmail.com), Software development consultant, Independent

27 Jun 2006

The recently released Google Web Toolkit (GWT) is a comprehensive set of APIs and tools that lets you create dynamic Web applications almost entirely in Java™ code. Philip McCarthy returns to his popular *Ajax for Java developers* series to show you what GWT can do and help you decide whether it's right for you. (*Note: You can now download an updated ZIP file containing the article source code.*)

More dW content related to: **google web toolkit form**

The GWT (see Resources) takes an unusual approach to Web application development. Rather than employing the normal separation of client-side and server-side codebases, GWT provides a Java API that lets you create component-based GUIs and then compile them for display in the user's Web browser. Using GWT is far closer to developing with Swing or SWT than the usual experience of Web application development, and it tries to abstract away the HTTP protocol and the HTML DOM model. Indeed, the fact that the application ends up being rendered in a Web browser feels almost incidental.

GWT achieves these feats through code generation, with GWT's compiler generating JavaScript from your client-side Java code. It supports a subset of the `java.lang` and `java.util` packages, along with the APIs that GWT itself provides. A compiled GWT application consists of fragments of HTML, XML, and JavaScript. However, these are pretty much indecipherable, and the compiled application is best regarded as a black box -- GWT's equivalent of Java bytecode.

In this article, I'll run through creating a simple GWT application to fetch a weather report from a remote Web API and display it in the browser. On the way, I'll briefly cover as many of GWT's capabilities as possible, and I'll mention some of the potential problems you'll come across.

Starting simple

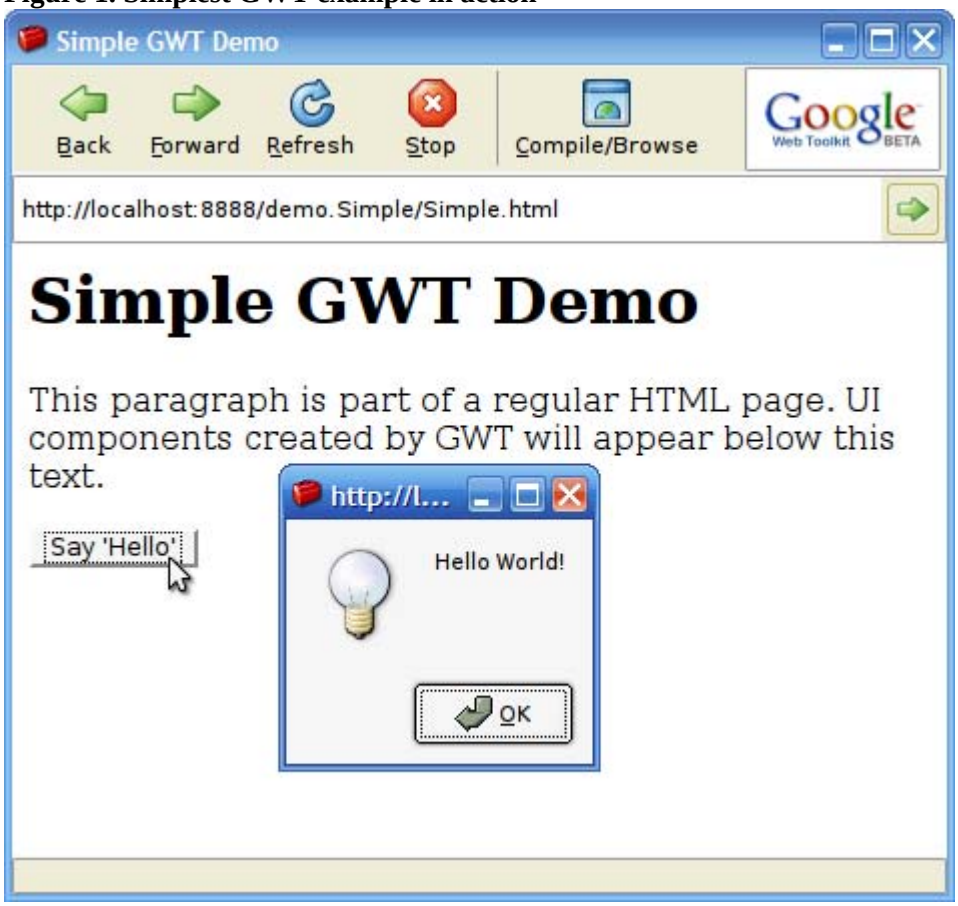
Listing 1 shows the Java source code of pretty much the simplest possible application you can make using GWT:

Listing 1. Simplest GWT example

```
public class Simple implements EntryPoint {

   public void onModuleLoad() {
     final Button button = new Button("Say 'Hello'");

     button.addClickListener(new ClickListener() {
        public void onClick(Widget sender) {
        Window.alert("Hello World!");
        }
     });

     RootPanel.get().add(button);
   }
}
```

This looks a lot like GUI code you might have written in Swing, AWT, or SWT. As you can probably guess, Listing 1 creates a button that displays a "Hello World!" message when you click it. The button is added to the `RootPanel`, a GWT wrapper around the body of an HTML page. Figure 1 shows the application in action, running inside GWT Shell. The Shell is a debugging hosting environment, incorporating a simple Web browser, that's included in the GWT SDK.

Figure 1. Simplest GWT example in action



Building the Weather Reporter application

I'm going to use GWT to create a simple Weather Reporter application. The application's GUI presents the user with an input box for entering a ZIP code and a choice of Celsius or Fahrenheit to represent temperatures. When the user clicks on a Submit button, the application uses Yahoo!'s free Weather API to obtain an RSS-formatted report for the chosen location. The HTML portion of this document is extracted and displayed to the user.

GWT applications are packaged as *modules* and must conform to a specific structure. A configuration file -- named *module-name*.gwt.xml -- defines the class that acts as the application's entry point and indicates whether resources should be inherited from other GWT modules. You must place the configuration file in the application's source package structure at the same level as a package named `client`, where all the client-side Java code resides, and a directory named public, which contains project Web resources such as images, CSS, and HTML. Finally, the public directory must include an HTML file with a `meta` tag containing the module's qualified name. GWT's run-time JavaScript library uses this file to initialize the application.

GWT's `applicationCreator` generates this basic structure for you, given the name of your entry-point class. So calling `applicationCreator developerworks.gwt.weather.client.Weather` generates a project outline I can use as a starting point for the Weather Reporter application. The source download for the application includes an Ant buildfile containing some useful targets for working with a GWT project conforming to this structure (see Download).

Developing the basic GUI

First, I'll develop the basic layout of the application's user-interface widgets, without adding any behavior. The superclass of pretty much everything you can render in a GWT UI is the `Widget` class. `Widgets` are always contained in `Panels`, which are themselves `Widgets` and therefore can be nested. Different types of panels offer different layout behaviors. So, a GWT `Panel` plays a similar role to that of a `Layout` in AWT/Swing or a `Box` in

XUL.

All widgets and panels must ultimately be attached to the Web page that hosts them. As you saw in Listing 1, you can attach them directly to the `RootPanel`. Alternatively, you can use `RootPanel` to obtain references to HTML elements identified by their IDs or classnames. In this case, I'll use two separate HTML `DIV` elements named `input-container` and `output-container`. The first contains the UI controls for the Weather Reporter; the second displays the weather report itself.

Listing 2 shows the code needed to set up the basic layout; it should be self-explanatory. The `HTML` widget is simply a container for HTML markup. This is where the HTML output from the Yahoo! weather feed is displayed. All of this code goes inside the `Weather` class's `onModuleLoad()` method, provided by the `EntryPoint` interface. This method is invoked when a Web page embedding the Weather module is loaded into a client Web browser.

Listing 2. Layout code for the Weather Reporter

```
public void onModuleLoad() {

   HorizontalPanel inputPanel = new HorizontalPanel();

   // Align child widgets along middle of panel
   inputPanel.setVerticalAlignment(HasVerticalAlignment.ALIGN_MIDDLE);

   Label lbl = new Label("5-digit zipcode: ");
   inputPanel.add(lbl);

   TextBox txBox = new TextBox();
   txBox.setVisibleLength(20);

   inputPanel.add(txBox);

   // Create radio button group to select units in C or F
   Panel radioPanel = new VerticalPanel();

   RadioButton ucRadio = new RadioButton("units", "Celsius");
   RadioButton ufRadio = new RadioButton("units", "Fahrenheit");

   // Default to Celsius
   ucRadio.setChecked(true);

   radioPanel.add(ucRadio);
   radioPanel.add(ufRadio);

   // Add radio buttons panel to inputs
   inputPanel.add(radioPanel);

   // Create Submit button
   Button btn = new Button("Submit");

   // Add button to inputs, aligned to bottom
   inputPanel.add(btn);
   inputPanel.setCellVerticalAlignment(btn,
      HasVerticalAlignment.ALIGN_BOTTOM);

   RootPanel.get("input-container").add(inputPanel);

   // Create widget for HTML output
   HTML weatherHtml = new HTML();

   RootPanel.get("output-container").add(weatherHtml);
}
```

Figure 2 shows the layout rendered in the GWT Shell:

Figure 2. Basic GUI layout

Adding styling with CSS

The rendered Web page is looking pretty dull, so it would benefit from some CSS styling rules. You can take a couple of approaches to styling a GWT application. First, by default, each widget has a CSS classname of the form *`project-widget`*. For example, `gwt-Button` and `gwt-RadioButton` are two of the core GWT widgets' classnames. Panels are generally implemented as a mess of nested tables and don't have default classnames.

The default classname-per-widget-type approach makes it easy to style widgets uniformly across your application. Of course, normal CSS selector rules apply, and you can exploit them to apply different styles to the same widget type depending on its context. For even more flexibility, you can replace or augment widgets' default classnames on an ad-hoc basis by calling their `setStyleName()` and `addStyleName()` methods.

Listing 3 combines these approaches to apply styles to the Weather Reporter application's input panel. The `weather-input-panel` classname is created in Weather.java with a call to `inputPanel.setStyleName("weather-input-panel");`.

Listing 3. Applying CSS styles to the Weather Reporter application's input panel

```
/* Style the input panel itself */
.weather-input-panel {
   background-color: #AACCFF;
   border: 2px solid #3366CC;
   font-weight: bold;
}

/* Apply padding to every element within the input panel */
.weather-input-panel * {
   padding: 3px;
}

/* Override the default button style */
.gwt-Button {
   background-color: #3366CC;
   color: white;
   font-weight: bold;
   border: 1px solid #AACCFF;
}

/* Apply a hover effect to the button */
.gwt-Button:hover {
   background-color: #FF0084;
}
```

Figure 3 shows the application again, with these styles in place:

Figure 3. Input panel with styles applied

Adding client-side behavior

Now that the basic layout and styling of the application is done, I can start applying some client-side behavior. You use the familiar Listener pattern to perform event handling in GWT. GWT provides `Listener` interfaces for mouse events, keyboard events, change events, and so forth, as well as several adapter and helper classes for added convenience.

Generally, you add event listeners using the anonymous inner-class idiom familiar to Swing programmers. However, the first parameter of all GWT's `Listener` methods is the event's sender, usually the widget that the user just interacted with. This means that you can attach the same `Listener` instance to multiple widgets where necessary; you can use the sender parameter to determine which of them fired the event.

Listing 4 shows the implementation of two event listeners in the Weather Reporter application. A click handler is added to the **Submit** button, and a keyhandler is added to the `TextBox`. Either clicking on the **Submit** button or pressing the Enter key when the `TextBox` has focus causes the associated handler to call the private `validateAndSubmit()` method. In addition to the code in Listing 4, `txBox` and `ucRadio` have become instance members of the `Weather` class so that they can be accessed from the validation method.

Listing 4. Adding client-side behavior

```
// Create Submit button, with click listener inner class attached
Button btn = new Button("Submit", new ClickListener() {

   public void onClick(Widget sender) {
      validateAndSubmit();
   }
});

// For usability, also submit data when the user hits Enter
// when the textbox has focus
txBox.addKeyboardListener(new KeyboardListenerAdapter(){

   public void onKeyPress(Widget sender, char keyCode, int modifiers) {

      // Check for Enter key
      if ((keyCode == 13) && (modifiers == 0)) {
         validateAndSubmit();
      }
   }
```

```
});
```

Listing 5 shows the implementation of the `validateAndSubmit()` method. It's fairly simple, relying on a `ZipCodeValidator` class that encapsulates the validation logic. If the user has not entered a valid five-digit ZIP code, `validateAndSubmit()` displays an error message in an alert box, expressed in the GWT world as a call to `Window.alert()`. If the ZIP code is valid, then both it and the user's choice of Celsius or Fahrenheit units are passed to the `fetchWeatherHtml()` method, which I'll get to a little later.

Listing 5. validateAndSubmit logic

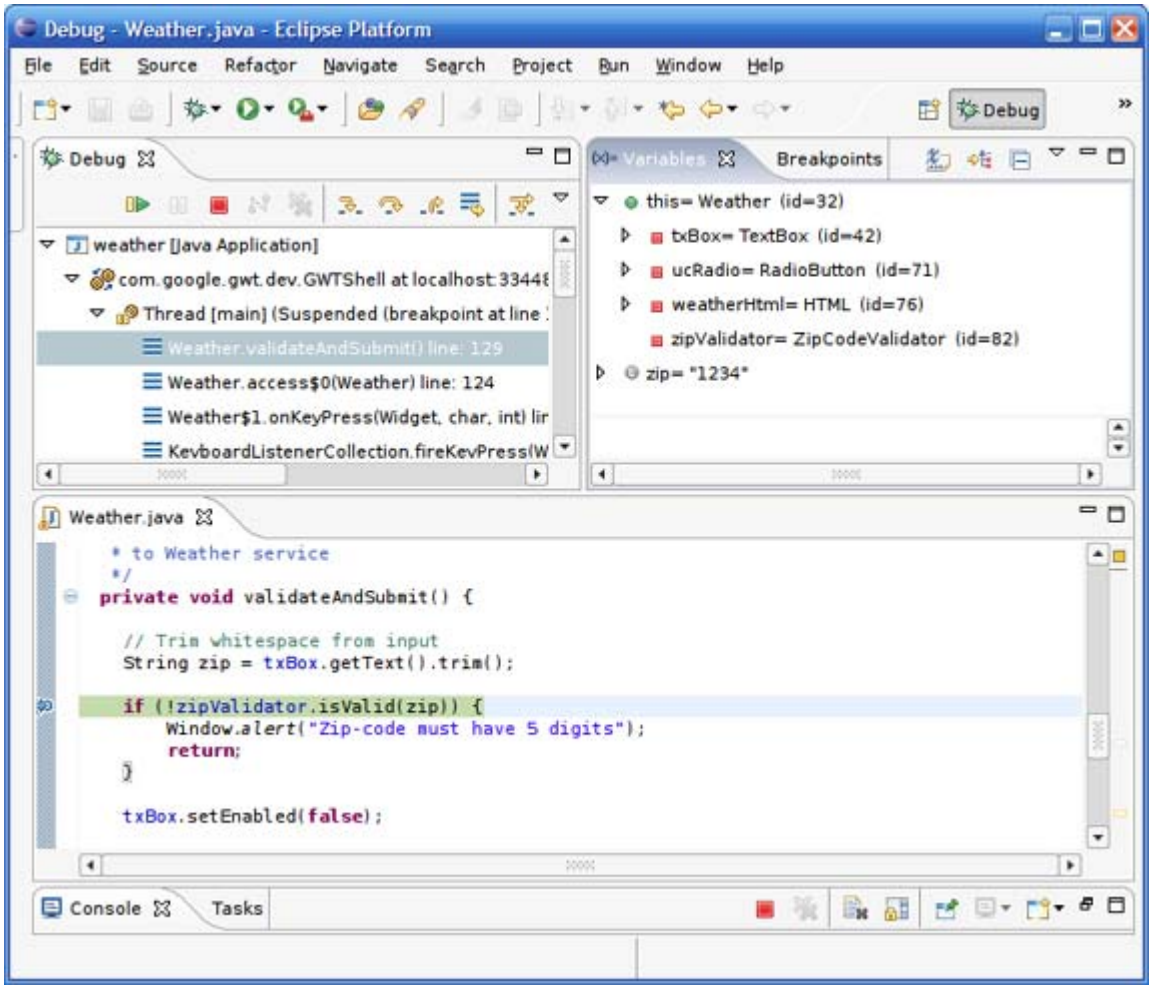
```
private void validateAndSubmit() {

   // Trim whitespace from input
   String zip = txBox.getText().trim();

   if (!zipValidator.isValid(zip)) {
     Window.alert("Zip-code must have 5 digits");
     return;
   }

   // Disable the TextBox
   txBox.setEnabled(false);

   // Get choice of celsius/fahrenheit
   boolean celsius = ucRadio.isChecked();
   fetchWeatherHtml(zip, celsius);
}
```

Client-side debugging with the GWT Shell

I'm going to sidetrack a little here to mention that the GWT Shell has JVM hooks that let you debug your client-side code in a Java IDE. You can interact with your Web UI and step through the Java code that represents the corresponding JavaScript executed on the client. This is an important ability because debugging the generated JavaScript on the client side is basically a nonstarter.

It's easy to configure an Eclipse debug task to launch the Shell via the `com.google.gwt.dev.GWTShell` class. Figure 4 shows Eclipse paused at a breakpoint in the `validateAndSubmit()` method, following a click on the **Submit** button:

Figure 4. Eclipse debugging client-side GWT code



Communicating with server-side components

Now the Weather Reporter application can collect and verify user input. The next step is to fetch data from the server. In normal Ajax development, this would entail calling a server-side resource directly from JavaScript and receiving data back encoded as JavaScript Object Notation (JSON) or XML. GWT abstracts this communication process behind its own remote procedure call (RPC) mechanism.

In GWT terminology, client code communicates with *services* running on the Web server. The RPC mechanism used to expose these services has similarities to the approach Java RMI uses. This means you only need to write the server-side implementation of your service and a couple of interfaces. Code generation and reflection take care of client stubs and server-side skeleton proxies.

Accordingly, the first step is to define an interface for the Weather Reporter service. This interface must extend the GWT `RemoteService` interface, and it contains the signatures of service methods that should be exposed to the GWT client code. Because RPC calls in GWT are between JavaScript code and Java code, GWT incorporates an object-serialization mechanism to mediate arguments and return values across the language divide (see the Serializable types sidebar to see what you can use).

With a service interface defined, the next step is to implement it in a class that extends GWT's `RemoteServiceServlet` class. As the name suggests, this is a specialization of the Java language's `HttpServlet`, so it can be hosted on any servlet container.

One GWT peculiarity worth mentioning here is that the service's remote interface must live in your application's `client` package because it needs to be incorporated in the JavaScript generation process. However, because the server-side implementation class references the remote interface, a Java compile-time dependency

Serializable types

A brief rundown of serializable types under GWT goes like this:

- Primitives (such as `int`) and the primitive wrapper classes (such as `Integer`) are serializable.
- `String` and `Date` are serializable.

now exists between server-side and client code. My solution to this is to place the remote interface into a `common` subpackage of `client`. I then include `common` in the Java build but exclude the rest of the `client` package. This prevents class files from being generated from client code that only needs to be converted to JavaScript. A more elegant solution might be to split package structure across two source directories for client-side and server-side code and to duplicate common classes into both directories.

Listing 6 shows the remote service interface used in the Weather Reporter application: `WeatherService`. It takes a ZIP code and a Celsius/Fahrenheit flag as input and returns a `String` containing an HTML weather description. Listing 6 also shows the outline of `YahooWeatherServiceImpl`, which uses the Yahoo! weather API to obtain an RSS weather feed for the given ZIP code and extracts an HTML description from it.

- Arrays of serializable types are themselves serializable.
- User defined classes are serializable if all of their nontransient members are serializable and they implement GWT's `IsSerializable` interface.
- `Collection` classes can be used in conjunction with Javadoc annotations that state the serializable type that they contain.

Because client code is limited to the small subset of Java classes implemented by GWT anyway, these serializable types provide fairly comprehensive coverage.

Listing 6. Remote WeatherService interface and partial implementation

```
public interface WeatherService extends RemoteService {

   /**
    * Return HTML description of weather
    * @param zip zipcode to fetch weather for
    * @param isCelsius true to fetch temperatures in celsius,
    * false for fahrenheit
    * @return HTML description of weather for zipcode area
    */
   public String getWeatherHtml(String zip, boolean isCelsius)
      throws WeatherException;
}

public class YahooWeatherServiceImpl extends RemoteServiceServlet
   implements WeatherService {

   /**
    * Return HTML description of weather
    * @param zip zipcode to fetch weather for
    * @param isCelsius true to fetch temperatures in celsius,
    * false for fahrenheit
    * @return HTML description of weather for zipcode area
    */
   public String getWeatherHtml(String zip, boolean isCelsius)
      throws WeatherException {

     // Clever programming goes here
   }
}
```

Things begin to diverge from the standard RMI approach at this point. Because Ajax calls from JavaScript are asynchronous, some extra work is needed to define an asynchronous interface that the client code uses to invoke the service. The asynchronous interface's method signatures differ from those of the remote interface, so GWT relies on Magical Coincidental Naming. In other words, no static compile-time relationship exists between the asynchronous interface and the remote interface, but GWT figures it out through naming conventions. Listing 7 shows the asynchronous interface for `WeatherService`:

Listing 7. Asynchronous interface for WeatherService

```
public interface WeatherServiceAsync {

   /**
    * Fetch HTML description of weather, pass to callback
    * @param zip zipcode to fetch weather for
    * @param isCelsius true to fetch temperatures in celsius,
    * false for fahrenheit
```

```
    * @param callback Weather HTML will be passed to this callback handler
    */
   public void getWeatherHtml(String zip, boolean isCelsius,
      AsyncCallback callback);
}
```

As you can see, the general idea is to create an interface called `MyServiceAsync` and to provide counterparts for each method signature, removing the return type and adding an extra parameter of type `AsyncCallback`. The asynchronous interface must also live in the same package as the remote interface. The `AsyncCallback` class has two methods: `onSuccess()` and `onFailure()`. If the call to the service is successful, then `onSuccess()` is invoked with the return value of the service call. If the remote call fails, `onFailure()` is invoked and passed the `Throwable` generated by the service to represent the cause of the failure.

Invoking the service from the client

With the `WeatherService` and its asynchronous interface in place, I can now modify the Weather Reporter client to invoke the service and handle its response. The first step is just boilerplate setup code: it creates an instance of `WeatherServiceAsync` for the Weather client to use by calling `GWT.create(WeatherService.class)` and downcasting the object it returns. Next, the `WeatherServiceAsync` must be cast to a `ServiceDefTarget` so that `setServiceEntryPoint()` can be called on it. `setServiceEntryPoint()` points the `WeatherServiceAsync` stub at the URL where its corresponding remote service implementation is deployed. Note that this is effectively hardcoded at compile time. Because this code becomes JavaScript deployed in a Web browser, there's no way to look up this URL from a properties file at run time. Obviously, this constrains the portability of a compiled GWT Web application.

Listing 8 shows the setup of the `WeatherServiceAsync` object and then gives the implementation of `fetchWeatherHtm()`, which I mentioned earlier (see Adding client-side behavior):

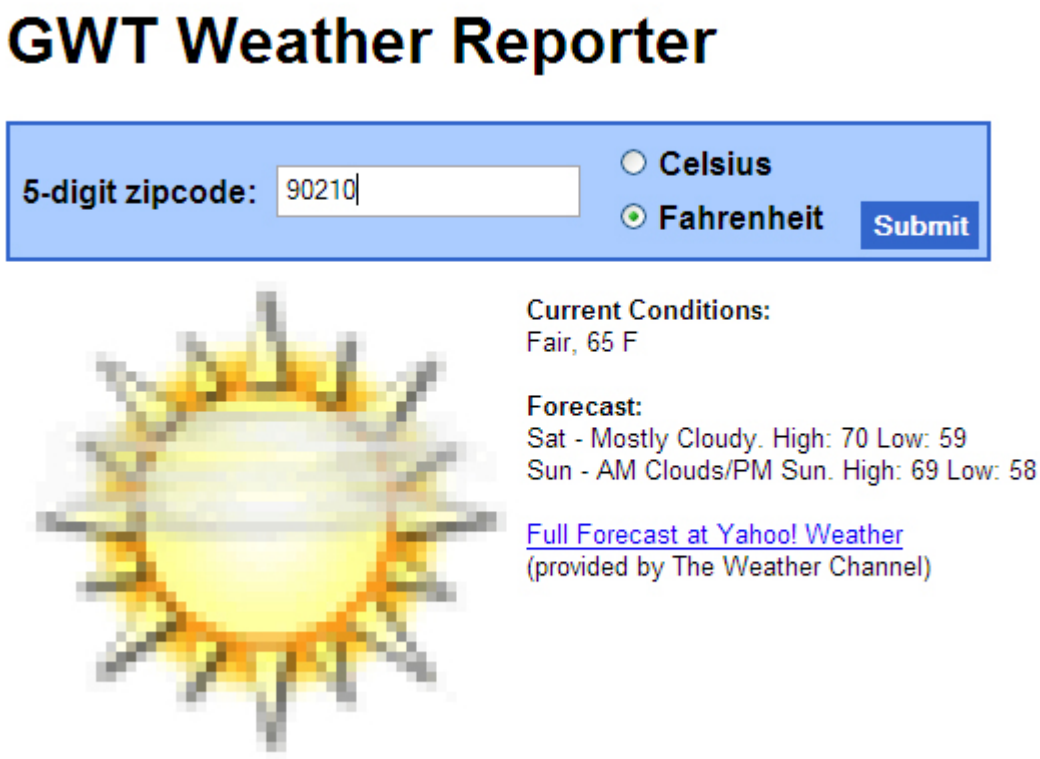
Listing 8. Using RPC to invoke a remote service

```
// Statically configure RPC service
private static WeatherServiceAsync ws =
   (WeatherServiceAsync) GWT.create(WeatherService.class);
static {
   ((ServiceDefTarget) ws).setServiceEntryPoint("ws");
}

/**
 * Asynchronously call the weather service and display results
 */
private void fetchWeatherHtml(String zip, boolean isCelsius) {

   // Hide existing weather report
   hideHtml();

   // Call remote service and define callback behavior
   ws.getWeatherHtml(zip, isCelsius, new AsyncCallback() {
      public void onSuccess(Object result) {

         String html = (String) result;

         // Show new weather report
         displayHtml(html);
      }

      public void onFailure(Throwable caught) {
         Window.alert("Error: " + caught.getMessage());
         txBox.setEnabled(true);
       }
   });
}
```

The actual call to the service's `getWeatherHtml()` is straightforward to implement, with an anonymous callback handler class simply passing the server's response on to a method that displays it.

Figure 5 shows the application in action, displaying a weather report that has been fetched from Yahoo!'s weather API:

Figure 5. The Weather Reporter application displaying a report fetched from Yahoo!

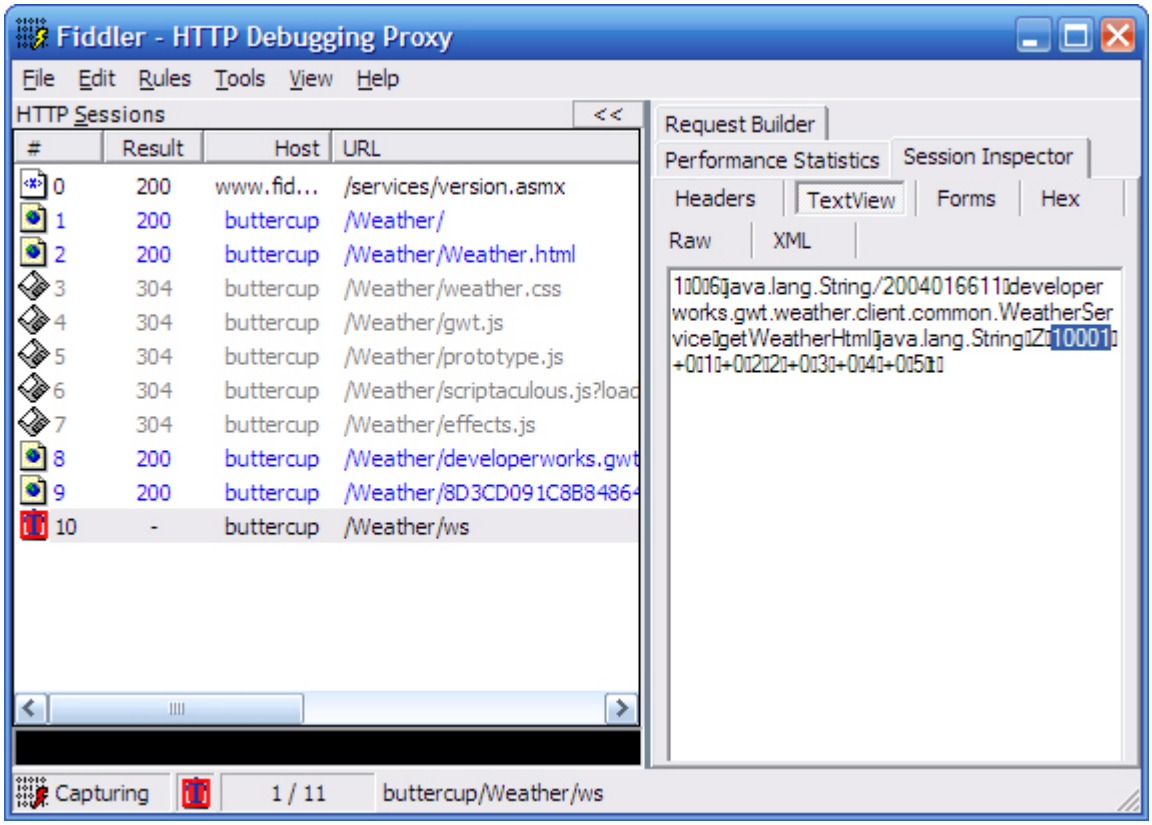


The need for server-side validation

GWT's conflation of client-side and server-side code is inherently dangerous. Because you program everything in the Java language, with GWT's abstraction concealing the client/server split, it's easy to be misled into thinking that your client-side code can be trusted at run time. This is a mistake. Any code that executes in a Web browser can be tampered with, or bypassed completely, by a malicious user. GWT provides a high level of obfuscation that mitigates the problem to a degree, but a secondary point of attack remains: any HTTP traffic that travels between your GWT client and its services.

Suppose I'm an attacker wishing to exploit weaknesses in the Weather Reporter application. Figure 6 shows Microsoft's Fiddler tool intercepting a request from the Weather Reporter client to the `WeatherService` running on the server. Once intercepted, Fiddler lets you alter any part of the request. The highlighted text shows that I've found where the ZIP code that I specified is encoded in the request. I can now alter this to anything I like, perhaps from "10001" to "XXXXX."

Figure 6. Using Fiddler to bypass client-side validation



Now, suppose some naive server-side code in the `YahooWeatherServiceImpl` calls `Integer.parseInt()` on the ZIP code. After all, the ZIP code must have passed the validation check that was incorporated into `Weather`'s `validateAndSubmit()` method, right? Well, as you've seen, this check has been subverted and now a `NumberFormatException` is thrown.

In this case, nothing terrible happens, and the attacker simply sees an error message in the client. However, the potential exists for a whole class of attacks against GWT applications dealing in more sensitive data. Imagine the ZIP code was instead a customer ID number in an order-tracking application. Intercepting and changing this value could expose sensitive financial information about other customers. Anyplace that a database query uses a value, the same approach allows the possibility of a SQL injection attack.

None of this should be rocket science to anyone who has worked with Ajax applications before. You simply need to double-check any input values by revalidating them on the server side. The key is remembering that some of the Java code you write in your GWT application is essentially untrustworthy at run time. This GWT cloud does have a silver lining, however. In the Weather Reporter application, I've already written a `ZipCodeValidator` for use on the client, so I can simply move it into my `client.common` package and reuse the same validation on the server side. Listing 9 shows this check incorporated into `YahooWeatherServiceImpl`:

Listing 9. ZipCodeValidator incorporated into YahooWeatherServiceImpl

```
public String getWeatherHtml(String zip, boolean isCelsius)
       throws WeatherException {

   if (!new ZipCodeValidator().isValid(zip)) {
      log.warn("Invalid zipcode: "+zip);
      throw new WeatherException("Zip-code must have 5 digits");
   }
```

Calling native JavaScript with JSNI

Visual-effects libraries are becoming increasingly popular in Web application development, whether their effects are used to provide subtle user-interaction cues or just to add polish. I'd like to add some eye-candy to the Weather

Reporter application. GWT doesn't provide this type of functionality, but its JavaScript Native Interface (JSNI) offers a solution. JSNI lets you make JavaScript calls directly from GWT client Java code. This means that I can exploit effects from the Scriptaculous library (see Resources) or from the Yahoo! User Interface library, for example.

JSNI uses a cunning combination of the Java language's `native` keyword and JavaScript embedded in a special comment block. It's probably best explained by example, so Listing 10 shows a method that invokes a given Scriptaculous effect on an `Element`:

Listing 10. Invoking Scriptaculous effects with JSNI

```
/**
 * Publishes HTML to the weather display pane
 */
private void displayHtml(String html) {
   weatherHtml.setHTML(html);
   applyEffect(weatherHtml.getElement(), "Appear");
}

/**
 * Applies a Scriptaculous effect to an element
 * @param element The element to reveal
 */
private native void applyEffect(Element element, String effectName) /*-{

   // Trigger named Scriptaculous effect
   $wnd.Effect[effectName](element);
}-*/;
```

This is perfectly valid Java code because the compiler sees only `private native void applyEffect(Element element, String effectName);`. GWT parses the contents of the comment block and outputs the JavaScript verbatim. GWT provides the `$wnd` and `$doc` variables to refer to the window and document objects. In this case, I'm simply accessing the top-level Scriptaculous `Effect` object and using JavaScript's square-bracket object-accessor syntax to invoke the named function specified by the caller. The `Element` type is a "magic" type provided by GWT that represents a `Widget`'s underlying HTML DOM element in both Java and JavaScript code. `String`s are one of the few other types that can be passed transparently between Java code and JavaScript via JSNI.

Now I have a weather report that fades in nicely when the data is returned from the server. The final touch is to re-enable the ZIP code `TextBox` when the effect has finished. Scriptaculous uses an asynchronous callback mechanism to notify listeners about the life cycle of effects. This is where things get a little more complex because I need to have JavaScript call back into my GWT client Java code. In JavaScript, you can invoke any function with an arbitrary number of arguments, so Java-style method overloading doesn't exist. This means that JSNI needs to use an unwieldy syntax to refer to Java methods to disambiguate possible overloads. The GWT documentation states this syntax as:

```
[instance-expr.]@class-name::method-name(param-signature)(arguments)
```

The `instance-expr.` part is optional because static methods are invoked without the need for an object reference. Again, it's easiest to see this illustrated by example, in Listing 11:

Listing 11. Calling back into Java code with JSNI

```
/**
 * Applies a Scriptaculous effect to an element
 * @param element The element to reveal
 */
private native void applyEffect(Element element, String effectName) /*-{

  // Keep reference to self for use inside closure
  var weather = this;

  // Trigger named Scriptaculous effect
  $wnd.Effect[effectName](element, {
```

```
     afterFinish : function () {

     // Make call back to Weather object
     weather.@developerworks.gwt.weather.client.Weather::effectFinished()();
     }
  });
}-*/;

/**
 * Callback triggered when a Scriptaculous effect finishes.
 * Re-enables the input textbox.
 */
private void effectFinished() {
  this.txBox.setEnabled(true);
  this.txBox.setFocus(true);
}
```

The `applyEffect()` method has been changed to pass an extra `afterFinish` argument to Scriptaculous. The value of `afterFinish` is an anonymous function that is invoked when the effect is done. This is somewhat similar to the anonymous inner-class idiom used by GWT's event handlers. The actual call back into Java code is made by specifying the instance of the `Weather` object to invoke the call on, then the fully qualified name of the `Weather` class, then the name of the function to invoke. The first empty pair of parentheses denotes that I want to call the method named `effectFinished()` that takes no arguments. The second set of parentheses calls the function.

The quirk here is that a local variable, `weather`, keeps a copy of the `this` reference. Because of the way JavaScript call semantics operate, the `this` variable inside the `afterFinish` function is actually a Scriptaculous object because Scriptaculous makes the function call. Making a copy of `this` outside the closure is a simple workaround.

Now that I've demonstrated some of JSNI's capabilities, I should point out that a much better way of integrating Scriptaculous into GWT is by wrapping Scriptaculous effects functionality up as a custom GWT widget. This is exactly what Alexei Sokolov has done in the GWT Component Library (see Resources).

Now that I'm all done with the Weather Reporter application, I'll examine some of the benefits and drawbacks of Web development with GWT.

Why use GWT?

Contrary to what you might expect, GWT applications are remarkably un-Weblike. GWT essentially exploits the browser as a run-time environment for lightweight GUI applications, and the result is much closer to what you might develop with Morfik, OpenLaszlo, or even Flash, than to normal Web applications. Accordingly, GWT is best suited to Web applications that can exist as a rich Ajax GUI on a single page. It's probably no coincidence that this characterizes some of Google's recent beta releases, such as its Calendar and Spreadsheets applications. These are great applications, but you can't solve all business scenarios using this approach. The majority of Web applications fit quite comfortably into the page-centric model, and Ajax lets richer interaction paradigms be employed where needed. GWT does not play nicely with traditional page-centric applications. Although it's possible to combine GWT widgets with normal HTML form inputs, the state of a GWT widget is fenced off from the rest of the page. For example, there's no straightforward way to submit the selected value from a GWT `Tree` widget as part of a regular form.

If you do decide to use GWT into a J2EE application environment, GWT's design should make integration relatively straightforward. In this scenario, GWT services should be thought of as similar to `Action`s in Struts -- a thin middle layer that simply proxies Web requests into business-logic invocations on the back end. Because GWT services are just HTTP servlets, they can be integrated easily into Struts or SpringMVC, for example, and placed behind authentication filters.

Licensing

GWT's run-time libraries are licensed under the Apache License 2.0, and you can use GWT freely to create commercial applications. However, the GWT toolchain is provided in binary-only form, and modifications are not permitted. This includes the

Java-to-JavaScript compiler. This means that any errors in your generated JavaScript are out of your control. A particular problem is GWT's reliance on user-agent detection: Each release of a new browser requires an update to the GWT toolkit to provide support.

GWT does have a couple of fairly significant flaws. First among them is its lack of provision for graceful degradation. Best practice in modern Web application development is to create pages that work without JavaScript, and then use it where available to embellish and add extra behavior. In GWT, if JavaScript isn't available, you won't get any UI at all. For certain classes of Web application, this is an immediate deal-breaker. Internationalization is also a major problem for GWT. Because GWT client Java classes run in the browser, they have no access to properties or resource bundles to grab localized strings at run time. A complex workaround is available that requires a subclass of each client-side class to be created for each locale (see Resources), but GWT's engineers are working on a more viable solution.

The case for code generation

Probably the most contentious issue in GWT's architecture is the switch to the Java language for client-side code. Some GWT proponents suggest that writing client-side code in the Java language is intrinsically preferable to writing JavaScript. This is not a view shared by all, and many JavaScript coders would be extremely reluctant to sacrifice their language's flexibility and expressiveness for the sometimes onerous grind of Java development. One situation where the substitution of Java code for JavaScript might be appealing is in a team that lacks experienced Web developers. However, if that team is moving into Ajax development, it might be better served by hiring skilled JavaScript programmers rather than relying on Java coders to produce obfuscated JavaScript at arm's length through a proprietary tool. Bugs are inevitably caused by leaks in the abstraction that GWT stretches over JavaScript, HTTP, and HTML, and inexperienced Web programmers have a hard time tracking them down. As developer and blogger Dimitri Glazkov puts it, "If you can't handle JavaScript, you shouldn't be writing code for Web applications. HTML, CSS, and JavaScript are the three prerequisites for this ride." (see Resources).

Some people argue that Java coding is inherently less error-prone than JavaScript programming, thanks to static typing and compile-time checks. This is a fairly fallacious argument. It's possible to write bad code in any language, and plenty of buggy Java applications are around to prove it. You're also dependent on GWT's code-generation being bug-free. However, offline syntax-checking and validation of client-side code can certainly be beneficial. It's available for JavaScript in the form of Douglas Crockford's JSLint (see Resources). GWT has the upper hand in terms of unit testing, though, providing JUnit integration for client-side code. Unit-testing support is an area where JavaScript is still sorely lacking.

In developing the Weather Reporter application, the most compelling case I found for client-side Java code was the ability to share the same validation class between both tiers. This obviously saves development effort. The same goes for any classes transferred over RPC; you only need to code them once and you can use them both client and server code. Unfortunately, the abstraction is leaky: in my ZIP code validator, for example, I would have liked to use regular expressions to perform the check. However, GWT doesn't implement the `String.match()` method. Even if it did, regular expressions in GWT have syntactical differences when deployed as client and server code. This is because of GWT's reliance on the host environment's underlying regexp mechanism and is an example of the trouble that imperfect abstractions can land you in.

One big win that GWT scores is its RPC mechanism and built-in serialization of objects between Java code and JavaScript. This removes a lot of the heavy lifting you see in the average Ajax application. It's not without precedent, though. If you want this functionality without the rest of GWT, then Direct Web Remoting (DWR), which offers RPC with object marshalling to and from Java code to JavaScript, is well worth considering (see Resources).

GWT also does a good job of abstracting away some of the low-level aspects of Ajax application development, such as cross-browser incompatibilities, the DOM event model, and making Ajax calls. But modern JavaScript toolkits such as the Yahoo! UI Library, Dojo, and MochiKit all provide a similar level of abstraction without needing to resort to code generation. Moreover, all of these toolkits are open source, so you can customize them to suit your needs or fix bugs that arise. This isn't possible with the black box of GWT (see the Licensing sidebar).

Conclusion

GWT is a comprehensive framework that provides a great deal of useful functionality. However, GWT is

something of an all-or-nothing approach, targeted at a relatively small niche in Web application development market. I hope this brief tour has given you a feel for GWT's capabilities and its limitations. Although it certainly won't suit everyone's needs, GWT remains a major engineering achievement and is worthy of serious consideration when you design your next Ajax application. GWT has more breadth and depth than I've been able to explore here, so do read Google's documentation for more or join the discussion on the GWT developer forum (see Resources).

Download

Description	Name	Size	Download method
GWT Weather Reporter application	j-ajax4-gwt-weather.zip	2.1KB	**HTTP**

Information about download methods Get Adobe® Reader®

Resources

Learn

- GWT: The GWT home page, with links to documentation and downloads.
- "Ajax for Java developers: Build dynamic Java applications" (Philip McCarthy, developerWorks, September 2005): First steps with Ajax.
- GWT Internationalization: GWT developer Scott Blum on the impractical approach currently needed for Internationalization of GWT applications.
- "GWT And Now Script#: The Agony of a Thousand Puppies": Web developer Dimitri Glazkov discusses the problem of GWT blurring Web abstraction layers.
- GWT on del.icio.us: Follow the buzz about GWT with its del.icio.us tag.
- Direct Web Remoting (DWR): DWR is a Java open source library providing an RPC mechanism from JavaScript to Java code.
- The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- GWT SDK: Download the Google Web Toolkit SDK.
- script.aculo.us: A JavaScript library for visual effects.
- GWT Component Library: A collection of third-party components created with GWT, including wrappers for Scriptaculous.
- JSLint: The JavaScript Verifier: JSLint provides syntax and structure validation for traditional JavaScript code.

Discuss

- GWT Developer Forum: Discussion group frequented by the developers of GWT, also serving as an ad-hoc source of GWT documentation.
- developerWorks blogs: Get involved in the developerWorks community.

About the author

Philip McCarthy is a software development consultant specializing in Java and Web technologies. He has recently been involved in digital media and telecoms projects at Hewlett Packard Labs and Orange and is currently working on financial software in the City of London.