DZone Refcardz

CONTENTS INCLUDE:

GWT

Style, Configuration and JSNI Reference

By Robert Hanson

ABOUT GOOGLE WEB TOOLKIT (GWT)

The Google Web Toolkit is a set of tools for writing JavaScript applications in Java. The cornerstone of the tool suite is a Java to JavaScript compiler that can not only compile Java down to JavaScript, but can also compress and optimize your code as well. GWT was released to the public in June of 2006 and has been an overwhelming success, boasting 1 million downloads in its first year.

The benefit of using GWT over some of the other JavaScript frameworks like JQuery or Ext-JS is the environment in which you code. GWT allows a Java developer to use the same tools they use today like Eclipse, Maven, and JUnit, making the transition from Java server-side development to GWT client-side development nearly seamless.

The GWT toolkit comes ready with its own widget library, internationalization tools, image bundling tools, tools for client-server communication, and many others. Since its launch GWT has become an open-source project, and although led by the GWT team at Google, many from the community have contributed patches to the toolkit. GWT boasts a thriving community, proving for a multitude of free widgets, integration tools, and utilities.

This reference card is a guide to everything that your IDE can't already tell you. Specifically, IDEs like Eclipse provide auto-completion capabilities for perusing a list of methods on an object, and often they also provide support for viewing javadocs in the IDE as well. This refcard is meant to supplement that capability by providing details that are only available in books and online documentation. Specifically this includes a CSS style for the widgets that ship with GWT, a reference for the JavaScript Native Interface, and a complete guide to the GWT module configuration file.

OFFICIAL GWT WEB SITES

WebSite: **http://code.google.com/webtoolkit/**

Blog: **http://googlewebtoolkit.blogspot.com/**

Forum: **http://groups.google.com/group/Google-Web-Toolkit**

Issue Tracker: **http://code.google.com/webtoolkit/issues/**

Articles: **http://code.google.com/webtoolkit/articles.html**

Examples: **http://code.google.com/webtoolkit/examples/**

Dev Guide: **http://code.google.com/webtoolkit/documentation/**

STYLING WIDGETS WITH CSS

Styling widgets is done using Cascading Style Sheets (CSS). CSS can be applied to the widgets in the application three different ways.

Adding a style element inside the <head> of the HTML page that is hosting the application.

```
<style type="text/css">
   /* CSS style rules*/
</style>
```

Adding a <link> element to the <head> of the HTML page, referencing an external CSS stylesheet.

```
<link type="text/css" rel="stylesheet"
   href="url-to-file.css" />
```

Adding a <stylesheet> element to the GWT project's module configuration, causing the stylesheet to be injected into the host HTML page.

```
<stylesheet src="url-to-file.css"/>
```

Most of the widgets that come with GWT have been pre-assigned CSS class names. For example, the Button widget uses the CSS class name gwt-Button. So you could set the width of all Button widgets in your application by using the following CSS rule. In order to reference a CSS class in a CSS rule you prefix the class name with a period ".".

```
.gwt-Button {
   width: 100px;
}
```

STYLING WIDGETS WITH CSS, *continued*

Altering Style Names

You can programmatically alter the CSS class name used on a widget by calling any of these methods.

widget.setStylePrimaryName("styleName")

In HTML you may provide any number of CSS class names on an element by separating them with spaces, like in the HTML snippet below.

```
<div class="style1 style2 style3"></div>
```

The primary style name in GWT is defined as the first style name in the class attribute. In the example provided, this would be the style "style1". Calling setStylePrimaryName() allows you to alter this first style.

widget.addStyleDependentName("styleName")

When you add a dependent style name, its name in the HTML is the primary name plus the depende nt name, separated with a dash ("-").

Example:

```
Button button = new Button();
button.setStylePrimaryName("foo");
button.addStyleDependentName("bar");
```

Result:

```
<BUTTON class="foo foo-bar"></BUTTON>
```

widget.setStyleName("styleName")

Using setStylename() will clear all current style names, including the primary style name, and adds the one provided.

widget.addStyleName("styleName")

Adds an additional style name to any existing style names.

Example:

```
Button button = new Button();
button.setStyleName("foo");
button.addStyleName("bar");
```

Result:

```
<BUTTON class="foo bar"></BUTTON>
```

widget.removeStyleName("styleName")

Allows you to remove an existing style name on a widget.

Default GWT Widget Style Names

Most of the widgets provided by the GWT library have pre-defined primary style names. The following is a list of the default names for each widget.

Widget	Default Name
Button	.gwt-Button { }
Checkbox	.gwt-CheckBox { }
DialogBox	.gwt-DialogBox { the box container } .gwt-DialogBox .Caption { the box caption }
DisclosurePanel	.gwt-DisclosurePanel { primary style } .gwt-DisclosurePanel-open { when open } .gwt-DisclosurePanel-closed { when closed } .header { the panel header area } .content { the panel content area }
HorizontalSplitPanel	.gwt-HorizontalSplitPanel { the panel } .gwt-HorizontalSplitPanel hsplitter { splitter }
HTML	.gwt-HTML { }
Hyperlink	.gwt-Hyperlink { }
Image	.gwt-Image { } ***Note:*** *Transformations between clipped and upclipped will result in the loss of any CSS style names that were set or added.*
Label	.gwt-Label { }
ListBox	.gwt-ListBox { }
MenuBar	.gwt-MenuBar { the menu bar itself } .gwt-MenuBar .gwt-MenuItem { menu items } .gwt-MenuBar .gwt-MenuItem-selected { selected menu items }
PasswordTextBox	.gwt-PasswordTextBox { primary style } .gwt-PasswordTextBox-readonly { dependent style set when the password text box is read-only }
PushButton	.gwt-PushButton-up {} .gwt-PushButton-down {} .gwt-PushButton-up-hovering {} .gwt-PushButton-down-hovering {} .gwt-PushButton-up-disabled {} .gwt-PushButton-down-disabled {} <any of the above> .html-face {}
RadioButton	.gwt-RadioButton { }
RichTextArea	.gwt-RichTextArea { }
StackPanel	.gwt-StackPanel { the panel itself } .gwt-StackPanel .gwt-StackPanelItem { unselected items } .gwt-StackPanel .gwt-StackPanelItem-selected { selected items }
SuggestBox	.gwt-SuggestBox { the suggest box itself } .gwt-SuggestBoxPopup { the suggestion popup } .gwt-SuggestBoxPopup .item { an unselected suggestion } .gwt-SuggestBoxPopup .item-selected { a selected suggestion }
TabBar	.gwt-TabBar { the tab bar itself } .gwt-TabBar .gwt-TabBarFirst { the left edge of the bar } .gwt-TabBar .gwt-TabBarRest { the right edge of the bar } .gwt-TabBar .gwt-TabBarItem { unselected tabs } .gwt-TabBar .gwt-TabBarItem-selected { additional style for selected tabs }
TabPanel	.gwt-TabPanel { the tab panel itself } .gwt-TabPanelBottom { the bottom section of the tab panel (the deck containing the widget) }
TextArea	.gwt-TextArea { primary style } .gwt-TextArea-readonly { dependent style set when the text area is read-only }
TextBox	.gwt-TextBox { primary style } .gwt-TextBox-readonly { dependent style set when the text box is read-only }
ToggleButton	.gwt- ToggleButton-up {} .gwt- ToggleButton-down {} .gwt- ToggleButton-up-hovering {} .gwt- ToggleButton-down-hovering {} .gwt- ToggleButton-up-disabled {} .gwt- ToggleButton-down-disabled {} <any of the above> .html-face {}
Tree	.gwt-Tree { the tree itself } .gwt-Tree .gwt-TreeItem { a tree item } .gwt-Tree .gwt-TreeItem-selected { a selected tree item }
VerticalSplitPanel	.gwt-VerticalSplitPanel { the panel itself } .gwt-VerticalSplitPanel vsplitter { the splitter }

GWT MODULE CONFIGURATION

A module in GWT is best described as a set of classes that are bound by a single module configuration file. The module configuration defines what classes are a part of the module, what other modules that the module depends on, as well as rules for deferred binding, resource injection, and everything else that the GWT compiler and shell needs to know about your module.

A module configuration file is located in the GWT project, with an extension of ".gwt.xml". The location on the classpath and the module configuration file name determine the full module name.

Module Name =
Java Package + Module File Name (-gwt.xml)

For example, if you have a module configuration file named Demo.gwt.xml, in the java package com.gwtsandbox.demo, the module name would be com.gwtsandbox.demo.Demo.

Hot Tip

The module configuration is only used at design and compile-time, it is not used at run-time. This is a common mistake for new GWT developers.

Simple Module Configuration

A simple module configuration must inherit the User module and specify a single entry point. The entry point is the class that implements the EntryPoint interface and acts as the starting point for the application.

```
<module>
   <inherits name='com.google.gwt.user.User' />
   <entry-point
      class='org.gwtsandbox.demo.client.Demo' />
</module>
```

From the basic module configuration you can build on it by adding additional elements to inherit additional modules, change the default source path, add servlet mappings, and add deferred binding rules.

Inheriting Modules

If your GWT project needs to reference external GWT modules you must explicitly inherit them in your module configuration. The core GWT libraries are split into several modules, each of which is listed here. You will always need to include the User module, and optionally one or more of the others.

GWT widgets and core utilities

```
<inherits name="com.google.gwt.user.User" />
```

RequestBuilder and associated classes

```
<inherits name="com.google.gwt.http.HTTP" />
```

Internationalization tools and date/number formatting

```
<inherits name="com.google.gwt.i18n.I18N" />
```

Tools for using RPC with JavaScript Object Notation

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

XML parser and associated classes

```
<inherits name="com.google.gwt.xml.XML" />
```

Source Path

The source path is a relative path name used to override the default location of the client-side Java source destined to be compiled into JavaScript. The source path you specify is appended to the path where the module configuration file resides, and you may specify multiple source paths.

```
<source path="path"/>
```

By default the source path is "client". So by way of example, if your GWT module configuration file is located at `com.example.MyApp.gwt.xml`, then the default path of "client" will dictate that your client-side Java source will be located in the Java package `com.example.client.*`, as well as all packages below this one.

All source code in the source path(s) must be able to be compiled to JavaScript with the GWT compiler. This implies that only classes from the JRE emulation library and GWT user library be used. For example, you can not include Java servlets or use the java.sql.* classes under this path.

Public Path

The public path is used to store non-Java files that need to be included in the application. This includes HTML files, JavaScript files, images, CSS, and anything else. By default this will be the "public" directory below where the module configuration is stored.

You can override the default by using the <public> tag in the module configuration.

```
<public path="path"/>
```

You may specify multiple public paths if required for your project.

Defining GWT-RPC Servlets

You can use the <servlet> tag in the module configuration to define your GWT-RPC servlets.

```
<servlet path="/path"
class="org.gwtsandbox.demo.server.Demo" />
```

The path specified should be absolute.

Hot Tip

These servlet mappings are for the benefit of hosted-mode use only, and does not imply that these mappings will be carried over to your production environment. For that you would set them up in the deployment descriptor, just as you would with any other servlet.

Resource Injection

You can have external JavaScript and CSS files automatically injected into the hosting web page. By injecting the resources you avoid the need to have the hosting HTML page explicitly include them with <link> and <script> tags. Resources loaded in this way will be loaded prior to the executing of the GWT application.

```
<script src="js-url"/>
<stylesheet src="css-url"/>
```

To inject a resource you can either place the JavaScript or CSS file into the public package of the GWT module (see above), referencing it with a relative path, or reference an external CSS file by using a full URL.

GWT MODULE CONFIGURATION, *continued*

Deferred Binding

In some cases you need to write low-level functionality that differs based on the client browser, or you need to trigger a generator to generate code at compile time. For these functions you use deferred binding. Deferred binding allows you to write code to an interface and have the concrete class determined at compile-time.

For example, you may be familiar with GWT's RPC mechanism. You use the GWT.create() method to return a concrete class that can serialize and send your data to the server.

```
MyServiceAsync svc =
   (MyServiceAsync) GWT.create(MyService.class);
```

When this code is compiled the compiler examines the argument passed to the create method, then attempts to match the target class to a set of rules that reside in the module configuration.

Using Generate-With to Trigger Generators

In this case the compiler rule is specified in the module com.google.gwt.user.RemoteService, which is inherited from your module when using GWT-RPC.

```
<generate-with
   class="com.google.gwt.user.rebind.rpc.
ServiceInterfaceProxyGenerator">
   <when-type-assignable
class="com.google.gwt.user.client.rpc.RemoteSer-
vice"/>
</generate-with>
```

This rule states that when the target class of the GWT.create() is assignable to RemoteService, that the generator ServiceInterfaceProxyGenerator is executed. The generator then creates the code and returns the name of the class that should be returned by the create method.

Using Replace-With to Trigger Class Replacement

The other use of deferred binding is to specify a alternate class to be returned by GWT.create() based on available properties. The most common use of this is to use an alternate class depending on the client browser. The property that can be examined to determine this is "user.agent".

For example, the DOM class in GWT is used to perform low-level functions, where the browser implementations can differ. In order to allow for different browsers the following rule can be found in the com.google.gwt.user.DOM module.

```
<replace-with
class="com.google.gwt.user.client.impl.DOMImplIE6">
<when-type-is
class="com.google.gwt.user.client.impl.DOMImpl"/>
<when-property-is name="user.agent" value="ie6"/>
</replace-with>
```

In the DOM class the following code is used to "create" the correct implementation of the DOM class.

```
DOMImpl impl = (DOMImpl) GWT.create(DOMImpl.class);
```

When the user.agent property is "ie6" the replace-with rule specified above will return a DOM implementation that is specifically designed for Internet Explorer.

Generate-With and Replace-With Expressions

The following expression tags can be used within generate-with and replace-with tags. When any of the expression tags within the rule returns a true value, the code generation or class replacement is performed.

```
<when-property-is
   name="prop-name" value="matched-value" />
```

Returns true when the value of the property matches the specified value. For details on setting properties, see the Setting Properties section.

```
<when-type-assignable class="assignable-type" />
```

The target class is tested to see if it can be assigned to the specified assignable type.

```
<when-type-is class="type" />
```

Similar to when-type-assignable, except that the class must be an exact match.

```
<all>, <any>, <none>
```

Use these expression tags to group other expression tags. The <all> tag implies that all of the tags contained within it must be true. The <any> tag requires only one of the containing expressions to be true. The <none> tag requires that all contained expression tags return false.

Setting Properties

Property names and their possible values can be defined in the module configuration.

```
<define-property name="prop-name" values="vals" />
```

Creates a new property and comma separated list of the possible values. For example, you could use the following to define a view property that could be used to define three unique view types for the application.

```
<extend-property name="prop-name" values="vals" />
```

Extends the possible values for a property that has already been defined.

```
<set-property name="prop-name" value="value" />
```

Sets the value of a defined property. The value must be one of the possible values as defined by the <define-property> tag or one of the extended values as defined by the <extended-property> tag.

```
<property-provider name="prop-name">
```

Property values can be set at run-time by supplying a property provider. The contents of the <property-provider> tag is a block of JavaScript that returns the property value. The JavaScript code must be placed in a CDATA block to avoid parsing errors.

```
<property-provider name="view"><![CDATA[
   var view = __gwt_getMetaProperty("view");

if (!__gwt_isKnownPropertyValue("view",  view)) {
     view = 'basic';
}
return view;
]]></property-provider>
```

The JavaScript block can utilize the __gwt_getMetaProperty() method to get the value of GWT property defined in a <meta> tag in the HTML page, and can use __gwt_isKnownProperty-Value() to test that it is an allowed value. Here's an example of using the HTML <meta> tag to set the view as "extended".

```
<meta name="gwt:property" content="view=extended" />
```

JAVASCRIPT NATIVE INTERFACE

The JSNI interface is used as a gateway between your Java and JavaScript code. It allows you to create Java methods that have JavaScript code in the body, and then to have the JavaScript code call Java methods.

JSNI Methods

Methods containing JavaScript must use the native modifier so that a Java compiler will not validate its contents. Methods in Java marked as native may not have a method body.

```
public native void doStuff ();
```

GWT introduces a special comment syntax that follows the Java native rule of having no method body, while allowing you to provide JavaScript code that can be picked up by the GWT compiler.

```
public native void doStuff () /*-{
    // JavaScript code goes here
}-*/;
```

JSNI Variables

Due to the way GWT applications load, the "window" and "document" JavaScript objects point to the wrong objects. GWT provides these special variables for use in JSNI methods.

Variable	Purpose
$wnd	Alias for the JavaScript "window" object
$doc	Alias for the JavaScript "document" object

```
public native void doStuff () /*-{
    $wnd.alert('Hello World');
}-*/;
```

Passing Values Between Java and JavaScript

You can pass both Java primitives and objects into a JSNI method. The following rules define how Java values are available in JSNI methods and visa versa.

Java type	Where created	JavaScript type
numeric primitive (byte, short, char, int, long, float, double)	Both	numeric value
boolean primitive	Both	boolean value
String	Both	string value
JavaScriptObject	Must be created in JSNI method	An object
Java array	Must be created in Java	Not directly usable, can be passed back into Java code
All other Java objects	Must be created in Java	Special, see below

JSNI methods that have Java objects passed to them may use a special syntax in order to call methods and access properties of the object.

Calling Java Methods from JavaScript

The following syntax is used to call a Java method from a JSNI method.

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

Calling Java Methods from JavaScript, continued

The **instance expression** is the name of the variable passed into the method, or "this" to refer to the class instance, or blank for calling static methods.

The **class name** is the fully qualified name, followed by double colons and the **method name**.

The **parameter signature** is a list of the parameter types in the method that you are calling. This is needed in order to distinguish between two Java methods that have the same name but different sets or parameters. The following table defines the codes used to specify the parameter type.

Type Code	Java Type
Z	Boolean
B	Byte
C	Char
S	Short
I	Int
J	Long
F	Float
D	Double
L *fully-qualified-class* ;	Java objects. Uses "/" to delimit parts of the package name, e.g. "Ljava/lang/String;"
[*type*	Int

Types are listed one after the other without spaces. For example, "ZLjava/lang/String;[F" defines the method signature (boolean, String, float[]).

The argument list is exactly that, a list of the arguments being passed to the method.

Below are examples of calling methods from JavaScript on well known Java types.

Calling instance method with no arguments:

```
public native long extractTime (Date date) /*-{
    return date.@java.util.Date::getTime()();
}-*/;
```

Calling instance method with a single argument:

```
public native void appendText (StringBuffer sb, String txt)
/*-{
sb.@java.lang.StringBuffer::append(Ljava/lang/String;)
(txt);
}-*/;
```

Calling static method with multiple arguments:

```
public native int maxValue (int x, int y) /*-{
    return @java.lang.Math::max(II)(x, y);
}-*/;
```

Accessing Java properties from JavaScript

Accessing properties of a Java object from JavaScript is similar to calling a Java method.

```
[instance-expr.]@class-name::field-name
```

The same value rules apply as calling methods (see Calling Java Methods from JavaScript). The instance expression is the variable name passed into the JSNI method, "this" for the instance of the class, or left off to access a static property. The class name is the fully qualified package and class, and is separated from the **field name** with two colons. On the following page are some examples.

JAVASCRIPT NATIVE INTERFACE, *continued*

Accessing Java properties from JavaScript, continued

Reading and writing to field on "this"

```
// int currentIndex;
var cur = this.@org.example.Demo::currentIndex;
this.@org.example.Demo::currentIndex = 0;
```

Reading and writing to field on object

```
// String welcomeMessage;
var msg = obj.@org.example.Demo::welcomeMessage;

obj.@org.example.Demo::welcomeMessage = "Hello";
```

Reading and writing to a static field

```
// boolean lastResult;
var last = @org.example.Demo::lastResult;
@org.example.Demo::lastResult = true;
```

Hot Tip

You can use JSNI to dynamically create JavaScript methods that can be used by external JavaScript code to make calls into your application. The following code creates a JavaScript method jsMethod(string) on startup in the browser that can be used to call the javaMethod(String) in the GWT application.

```
public void onModuleLoad () {
        createJsMethod(this);
    }
    private native void createJsMethod (Main obj) /*-{
        $wnd.jsMethod = function (s) {
            return
obj.@com.getsandbox.demo.client.Main::javaMethod(Ljava/lang/
String;)(s);
        };
    }-*/;
    public String javaMethod (String in) {
        return "I got your message: " + in;
    }
    <button onclick="alert(jsMethod('Hello'))">Say Hello</button>
```

ABOUT THE AUTHOR

Robert Hanson

Robert Hanson is an Applications Manager for Quality Technology Services, a company providing hosting, managed services, and application development. He is passionate about programming and architecture, and has developed opensource software for both the Perl and Java communities. His latest venture is the GWT Widget Library, a set of tools for use with the Google Web Toolkit, a toolkit allowing you to write JavaScript applications in Java. In 2007 Robert co-authored the book *GWT in Action* with Adam Tacy, providing instruction on using the Google toolkit. Robert provides tutorials and random thoughts about the craft on his blog found at roberthanson.blogspot.com.

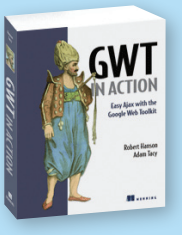
Publications

- *GWT in Action* with Adam Tacy

Blog

Roberthanson.blogspot.com

RECOMMENDED BOOK



GWT in Action shows you how to set up your development environment, use and create widgets, communicate with the server, and much more. Readers will follow an example running throughout the book and quickly master the basics of GWT: widgets, panels, and event handling.

BUY NOW

books.dzone.com/books/gwt-in-action

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices

Available:

Essential Ruby
Essential MySQL
JUnit and EasyMock
Getting Started with MyEclipse
Spring Annotations
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom

Visit refcardz.com for a complete listing of available Refcardz.

Design Patterns
Published June 2008

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-07-3
ISBN-10: 1-934238-07-4
50795
9 781934 238073
$7.95