Using Google Web Toolkit (GWT) for Developing AJAX-Based Web Applications for the SAP NetWeaver J2EE Framework

Applies to:
SAP NetWeaver J2EE Stack, Knowledge Management and Collaboration platform

Summary
This article summarizes steps necessary for the development of Ajax-based web applications using the Google Web Toolkit (GWT) within SAP NetWeaver J2EE framework.

Author(s): Sergey Zavarzin (sergey.zavarzin@surasystems.com)
Company: SuraSystems GmbH
Created on: 24 January 2007

Author Bio
Sergey is a Java developer with SuraSystems GmbH, specializing on SAP NetWeaver.
Table of Contents

Introduction ...................................................................................................................................... 2
The Step-by-Step Solution............................................................................................................... 3
   Creating J2EE Applications with GWT Framework Support......................................................... 3
      Creating J2EE Web Module ........................................................................................................ 3
      Compilation of the GWT Package Layout ............................................................................. 10
   Enterprise Application Wrapper................................................................................................. 11
   Test of Resulting Application..................................................................................................... 13
   Creating a J2EE Server Component Library............................................................................. 13
   Using GWT Library within SAP NetWeaver J2EE Application ................................................ 18
Disclaimer and Liability Notice....................................................................................................... 21

Introduction

Google Web Toolkit (GWT) is an open source Java software development framework that makes writing
AJAX applications like Google Maps and Gmail. GWT allows writing your front end in the Java programming
language and the GWT compiler converts your Java classes to browser-compliant JavaScript and HTML.
Here's the GWT development cycle:

- Use your favorite Java IDE to write and debug an application in the Java language, using as many (or
  as few) GWT libraries as you find useful.

- Use GWT's Java-to-JavaScript compiler to distill your application into a set of JavaScript and HTML
  files that you can serve with any web server.

In this document, we focus on the procedure for establishing a connection with KMC platform in the J2EE
environment and creating Ajax-based web application working with Knowledge Management.
The Step-by-Step Solution

In the first section below we create a simple SAP NetWeaver J2EE application that uses GWT library and in the second section we create a J2EE server component library for sharing the GWT library within SAP NetWeaver J2EE environment.

Creating J2EE Applications with GWT Framework Support

Developed J2EE application will use GWT library for client UI creation and get some data from KMC platform to display.

Creating J2EE Web Module

In J2EE Development Perspective for NWDS, create web module project `com.surasystems.kmclient`:

Create new packages: `com.surasystems.kmclient.client` for client UI module and `com.surasystems.kmclient.server` for server-side service for work with KM.

For working with GWT framework we have to create a project with hard structure of packages to translate our Java code to JavaScript with GWT compiler.

If you are starting a GWT project from scratch, you should use the standard GWT package layout, which makes it easy to differentiate client-side code from server-side code.

The standard package layout would look like this:
com/surasystems/kmclient The project root package contains module XML files
com/surasystems/kmclient/client Client-side source files and subpackages
com/surasystems/kmclient/server Server-side code and subpackages
com/surasystems/kmclient/public Static resources that can be served publicly

Individual units of GWT configuration are XML files called modules. A module bundles together all the configuration settings that your GWT project needs, namely:

- Inherited modules
- An entry point application class name; these are optional, although any module referred to in HTML must have at least one entry-point class specified
- Source path entries
- Public path entries
- Deferred binding rules, including property providers and class generators

Modules may appear in any package in your classpath, although it is strongly recommended that they appear in the root package of a standard GWT project layout.

GWT Modules are defined in XML files whose file extension is .gwt.xml. Module XML files should reside in your project's root package.

In this case KmClient.gwt.xml should look as follows:

```xml
<module>
  <inherits name='com.google.gwt.user.User'/>
  <entry-point class='com.surasystems.kmclient.client.KmClient'/>
  <stylesheet src="KmClient.css" />
</module>
```

The folder com/surasystems/kmclient/public contains two files: kmclient.html (entry-point for client-side web interface) and KmClient.css (for CSS styles describing).

We'll use a standard html file for GWT project as following:

```html
<html>
  <head>
    <title>Wrapper HTML for Client</title>
    <meta name='gwt:module' content='com.surasystems.kmclient.KmClient'/>
  </head>
</html>
```
We have only to set up meta tag "gwt:module" and link tag for style sheet file as shown above.

Then a new Java classes should be added with following code:

```java
import com.google.gwt.core.client EntryPoint;
import com.google.gwt.user.client.rpc AsyncCallback;
import com.google.gwt.user.client.ui ClickListener;
import com.google.gwt.user.client.ui Grid;
import com.google.gwt.user.client.ui ListBox;
import com.google.gwt.user.client.ui RootPanel;
import com.google.gwt.user.client.ui TextArea;
import com.google.gwt.user.client.ui Widget;

public class KmClient implements EntryPoint {

    private final IKmServiceAsync kmService;

    public KmClient() {
        // Getting a service
        kmService = IKmService.Util.getInstance();
    }

    final static int VISIBLE_ROWS = 20;

    private void changeList(ListBox list) {
    }

    public void onModuleLoad() {
        RootPanel rootPanel = RootPanel.get();
        rootPanel.setStyleName("");

        final Grid grid = new Grid(2, 2);
```
final TextArea textArea = new TextArea();
textArea.setWidth("300px");
textArea.setVisibleLines(VISIBLE_ROWS);
textArea.setEnabled(false);

final ListBox list = new ListBox();
list.addItem("/documents/Folder1/");
list.addItem("/documents/Folder1/Folder2/");
list.addItem("---");
list.setVisibleItemCount(VISIBLE_ROWS);
list.setWidth("300px");
list.addClickListener(new ClickListener() {
    public void onClick(Widget sender) {
        String selected = list.getItemText(list.getSelectedIndex());
        kmService.getCollection(selected, new AsyncCallback() {
            public void onFailure(Throwable caught) {
                textArea.setText("Error: Can't get data. ");
            }
            public void onSuccess(Object result) {
                list.clear();
                String[] data = result.toString().split(",");
                for (int i = 0; i < data.length; i++) {
                    list.addItem(data[i]);
                }
                list.setVisibleItemCount(VISIBLE_ROWS);
            }
        });
        kmService.getResourceProperties(selected, new AsyncCallback() {
            public void onFailure(Throwable caught) {
                textArea.setText("Error: Can't get data. ");
            }
            public void onSuccess(Object result) {
                textArea.setText(result.toString());
            }
        });
    }
});

grid.setWidget(0, 0, list);
grid.setWidget(0, 1, textArea);
rootPanel.add(grid);

The KmClient class is entry point realizing client-side web interface.

The GWT framework uses mechanism for interacting with a server across a network known as remote procedure call (RPC), also sometimes referred to as a server call. GWT RPC makes it easy for the client and server to pass Java objects back and forth over HTTP.

For PRC mechanism using we have to realize two following classes. The first class defines client-side Java interface for remote server-side service and the second one specifies an asynchronous interface based on our original service interface.

import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.ServiceDefTarget;

public interface IKmService extends RemoteService {
    /**
     * Utility class for simplifying access to the instance of async service.
     */
    public static class Util {
        private static IKmServiceAsync instance;
        public static IKmServiceAsync getInstance() {
            if (instance == null) {
                instance = (IKmServiceAsync) GWT.create(IKmService.class);
                ServiceDefTarget target = (ServiceDefTarget) instance;
                target.setServiceEntryPoint(GWT.getModuleBaseURL() + 
"/kmservice");
            }
            return instance;
        }
    }

    public String getResourceProperties(String dir);
    public String getResourceName(String dir);
    public String getCollection(String dir);
}

package com.surasystems.kmclient.client;
import com.google.gwt.user.client.rpc.AsyncCallback;
public interface IKmServiceAsync {
    public void getResourceProperties(String dir, AsyncCallback callback);
    public void getResourceName(String dir, AsyncCallback callback);
    public void getCollection(String dir, AsyncCallback callback);
}

The relationship between a service interface and its asynchronous counterpart is straightforward:

- If a service interface is called com.surasystems.kmclient.IKmService, then the asynchronous interface must be called com.surasystems.kmclient.IKmServiceAsync. The asynchronous interface must be in the same package and have the same name, but with the suffix Async.

- For each method in your service interface, for example:

```java
public String getResourceProperties(String dir);
```

an asynchronous sibling method should be defined that looks like this:

```java
public void getResourceProperties(String dir, AsyncCallback callback);
```

Then we should implement server-side service for work with KM resources. This service simply implements three methods for getting KM resource, collection and its properties. It should extend RemoteServiceServlet class and implement IKmService interface.
package com.surasystems.kmclient.server;

import java.io.IOException;
import java.security.AccessController;
import java.security.PrivilegedExceptionAction;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.sap.security.api.IUserFactory;
import com.sap.security.api.UMException;
import com.sap.security.api.UMFactory;
import com.sapportals.wcm.WcmException;
import com.sapportals.wcm.repository.AccessDeniedException;
import com.sapportals.wcm.repository.IPropertyIterator;
import com.sapportals.wcm.repository.IPropertyMap;
import com.sapportals.wcm.repository.IPropertyName;
import com.sapportals.wcm.repository.IResource;
import com.sapportals.wcm.repository.IResourceContext;
import com.sapportals.wcm.repository.IResourceFactory;
import com.sapportals.wcm.repository.IResourceList;
import com.sapportals.wcm.repository.IResourceListIterator;
import com.sapportals.wcm.util.uri.RID;
import com.surasystems.kmclient.client.IKmService;

public class KmService extends RemoteServiceServlet implements IKmService {

    /**
     * Get the resource
     * @param dir Absolute path for resource
     */
    public IResource getResource(String dir) {
        try {
            IResourceFactory resFactory = null;
            IResource resource = null;
            resFactory = ResourceFactory.getInstance();
            Object serviceContext = null;
            serviceContext = AccessController.doPrivileged(new PrivilegedExceptionAction() {
                public Object run() throws WcmException {
                    return ResourceFactory.getInstance().getServiceContext("ice_service");
                }
            });
            IResourceContext resContext = (IResourceContext) serviceContext;
            RID rid = RID.getRID(dir);
            resource = resFactory.getResource(rid, resContext);
            return resource;
        } catch (Exception e) { return null; }
    }

    public String getResourceProperties(IResource resource) {
        StringBuffer s = new StringBuffer();
        IPropertyMap pm;
        try {
            IPropertyName name = new PropertyName("http://sapportals.com/xmlns/cm",
                    "createdby");
            IProperty property = resource.getProperty(name);
            s.append("parameter")
        }
    }
}
"Access RID: " + resource.getAccessRID().toString() + "\n";

IUserFactory userFactory = UMFactory.getUserFactory();
String userName =
    userFactory
        .getUserByLogonID(property.getValueAsString())
        .getDisplayName();

s.append("Created By: " + userName + "\n");

pm = resource.getProperties();
IPropertyIterator ipm = pm.iterator();
while (ipm.hasNext()) {
    IProperty prop = ipm.next();
    s.append(prop.getPropertyName().getName() + "\n");
    s.append(prop.getValueAsString() + "\n");
}

return s.toString();
}

/**
 * llection list
 * @param dir
 * @return
 */

public String getCollection(String startDir) {

String listFolderNames = "";
IResource res;
ICollection collection;

res = getResource(startDir);
IW adopting collection;

try {
    listFolderNames += res.getParentCollection().getRID().getPath() + ";----------------------,;

    int size = collection.getChildren().size();
    IResourceList rl;
    rl = collection.getChildren();

    IResourceListIterator iterator = rl.listIterator();
    while (iterator.hasNext()) {
        listFolderNames += iterator.next().getRID().getPath() + ",";
    }
}

} catch (AccessDeniedException e) {
    e.printStackTrace();
} catch (ResourceException e) {
    e.printStackTrace();
}

listFolderNames =
    listFolderNames.substring(0, listFolderNames.length() - 1);

return listFolderNames;
}

public String getResourceProperties(String dir) {
    return getResourceProperties(getResource(dir));
}

public String getResourceName(String dir) {
    return getResource(dir).getDisplayName();
}
public static IResource resolveIfLink(IResource res) {
  try {
    if (res.getLinkType() == LinkType.NONE) {
      return res;
    } else {
      return res.getTargetResource();
    }
  }
  catch (Exception e) {
    return res;
  }
}

Resulting web.xml file for this web module should look as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc./DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>WEB APP</display-name>
  <description>WEB APP description</description>
  <servlet>
    <servlet-name>KmService</servlet-name>
    <servlet-class>com.surasystems.kmclient.server.KmService</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>KmService</servlet-name>
    <url-pattern>com.surasystems.kmclient.KmClient/kmservice</url-pattern>
  </servlet-mapping>
</web-app>
```

Compilation of the GWT Package Layout

The GWT compiler converts your Java classes to browser-compliant JavaScript and HTML. For compilation you can use the following command file that should be located in your project directory and have to be corrected for gwt-user.jar and gwt-dev-windows.jar finding, for example:

```
@java -cp "%~dp0\source;%~dp0\bin;C:\libs\gwt-windows-1.2.22\gwt-user.jar;C:\libs\gwt-windows-1.2.22\gwt-dev-windows.jar"
com.google.gwt.dev.GWTCompiler -out "%~dp0\webContent" -style "DETAILED" %*
com.surasystems.kmclient.KmClient
```

As a result of compilation you get a set of html files in output directory `webContent\com.surasystems.kmclient.KmClient` located in your project directory.

For using GWT library classes in run-time you should release your web module with its library. For that, place gwt-servlet.jar file in WEB-INF\lib directory of your project.
The next step is to build web module archive file (.war).

Enterprise Application Wrapper

In order to deploy the web module you have to wrap your created web module project with enterprise application (com.surasystems.kmclient.ear for instance).

You then can simply add your web application to enterprise application:
Since your server-side service uses SAP portal libraries for work with KM resources you should define references in `application-j2ee-engine.xml` file of enterprise application.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application-j2ee-engine SYSTEM "application-j2ee-engine.dtd">
<application-j2ee-engine>
  <reference reference-type="weak">
    <reference-target provider-name="sap.com" target-type="application">com.sap.km.application</reference-target>
  </reference>
</application-j2ee-engine>
```

The `application.xml` file should look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "/Sun Microsystems, Inc./DTD J2EE Application 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
<application>
  <display-name>com.surasystems.kmclient.ear</display-name>
  <description>EAR description</description>
  <module>
    <web>
      <web-uri>com.surasystems.kmclient.war</web-uri>
      <context-root>com.surasystems.kmclient</context-root>
    </web>
  </module>
</application>
```

Now all is ready to build and deploy our enterprise application:
Test of Resulting Application

To test the obtained J2EE application with client Ajax-based UI the following link should be used

This application simply read folder content in KM and gets resource properties using asynchronous calls from client side without any additional page reloading.

Creating a J2EE Server Component Library

For sharing GWT Library within SAP NetWeaver J2EE platform you should create a new J2EE Server Component -> Library.
Then you should define name of new created library, for example com.surasystems.kmclient.lib, as shown below.
Now you can add `gwt-library.jar` to this project:

Also you need to define “Provider Name”, “Component Name” and “Display Name” for new library in file `provider.xml`:
Moreover, you should add a reference to `gwt-library.jar` file in your library project:

Finally, since GWT library uses `javax.servlet.*` packages you should add references to this library within SAP J2EE environment:
Resulting provider.xml file should look as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE provider-descriptor SYSTEM "library.provider.dtd">
<provider-descriptor>
  <display-name>GWT Library</display-name>
  <component-name>com.surasystems.kmclient.lib</component-name>
  <major-version>6</major-version>
  <minor-version>40</minor-version>
  <micro-version>0</micro-version>
  <provider-name>surasystems.com</provider-name>
  <references>
    <reference
      provider-name="sap.com"
      strength="weak"
      type="library">servlet</reference>
  </references>
  <jars>
    <jar-name>gwt-servlet.jar</jar-name>
  </jars>
</provider-descriptor>
```

Then, you can build library archive file to deploy:
You can check the GWT Library configuration with Visual Administrator environment by expanding Cluster → Server → Libraries → GWT Library:

Using GWT Library within SAP NetWeaver J2EE Application

To use GWT Library as J2EE server component library from your J2EE application we should adjust application-j2ee-engine.xml file. For that the reference to library you created earlier should be added:
And resulting `application-j2ee-engine.xml` should look as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application-j2ee-engine SYSTEM "application-j2ee-engine.dtd">
<application-j2ee-engine>
  <reference reference-type="weak">
    <reference-target provider-name="sap.com" target-type="application">com.sap.km.application</reference-target>
  </reference>
  <reference reference-type="weak">
    <reference-target provider-name="surasystems.com" target-type="library">com.surasystems.kmclient.lib</reference-target>
  </reference>
  <fail-over-enable mode="disable"/>
</application-j2ee-engine>
```

Now you should build application archive and deploy it again.
You can check all references for your J2EE server component library with Visual Administrator environment by expanding **Cluster → Server → Services → ClassLoader Viewer and Runtime tab → surasystems.com~com.surasystems.kmclient.lib**
Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.