EasyBeans HowTo

Florent BENOIT, ObjectWeb consortium

Table of Contents

List of Figures

Chapter 1. GWT and EJB3 with EasyBeans

The GWT (Google Web Toolkit [http://code.google.com/webtoolkit/]) will be used as the framework for producing clients using AJAX [http://en.wikipedia.org/wiki/AJAX] technique while EJB3 will be on the server side in order to provide some services (like persistence).

EJB3 can ease the developers for the persistence part (with Java Persistence API) or for the business part.

The source of an application using EasyBeans and GWT can be browse on the Fisheye Sudoku EasyBeans site [http://fisheye.easybeans.org/viewrep/EasyBeans/trunk/easybeans-sudoku]. The application is a Sudoku Game and this howto use some code of this application.

Figure 1.1. Sudoku Game demo using GWT and EasyBeans

1.1. Requirements

1.1.1. JDK

A JDK 5.0 is required to use EasyBeans.

Tip

GWT only support JDK 1.4 functions. So classes that will be transformed by GWT into JavaScript shouldn't use JDK 5 features (like annotations). Other classes can use the new features (like the EJB3).

1.1.2. Eclipse

GWT can be used without Eclipse but in this HowTo, Eclipse [http://www.eclipse.org/downloads/] is used.

1.1.3. EasyBeans

This HowTo is done by using the EasyBeans Tomcat package. Note that it works with the JOnAS package too.

EasyBeans can be downloaded from EasyBeans Web Site [http://www.easybeans.org].

1.1.4. Tomcat

Tomcat 5.5.17 was used. It should work with other versions. Download from Tomcat site [http://tomcat.apache.org/download-55.cgi].

1.1.5. Google Web Toolkit

Toolkit can be downloaded from the Google Web Toolkit download page. [http://code.google.com/webtoolkit/download.html]

1.2. Starting with GWT

1.2.1. Creating an Eclipse Project

GWT comes with an eclipse project creator tool. The project will have the name Sudoku and will be created in the $HOME/workspace directory.

```
$ cd gwt-linux-1.0.21
$ ./projectCreator -eclipse Sudoku -out $HOME/workspace/sudoku
Created directory $HOME/workspace/sudoku/src
Created file $HOME/workspace/sudoku/.project
Created file $HOME/workspace/sudoku/.classpath
```

1.2.2. Adding a GWT Application to the Eclipse project

A GWT application needs to be added to the eclipse project. The name of the class to generate is given to the application creator tool.

Some files will be generated :

- The `Sudoku.gwt.xml` file which describes the module with entry point. This file will be also used to add the mapping for the remote service using EasyBeans.
- The `Sudoku.html` file which is a skeleton page which call the widget's examples.
- The `Sudoku.java` file which is the class responsible to load widgets (which implements the `EntryPoint` interface).
- Then, there are 3 scripts that allow to launch or compile files. Note that compile script will be replaced by an ant task in this HowTo.

```
$ ./applicationCreator -out $HOME/workspace/sudoku -eclipse -out
org.objectweb.easybeans.demo.sudoku.web.client.Sudoku
Created directory $HOME/workspace/sudoku/src/org/objectweb/easybeans/demo/sudoku/web
Created directory $HOME/workspace/sudoku/src/org/objectweb/easybeans/demo/sudoku/web/client
Created directory $HOME/workspace/sudoku/src/org/objectweb/easybeans/demo/sudoku/web/public
Created file $HOME/workspace/sudoku/src/org/objectweb/easybeans/demo/sudoku/web/Sudoku.gwt.xml
Created file
$HOME//workspace/sudoku/src/org/objectweb/easybeans/demo/sudoku/web/public/Sudoku.html
Created file
$HOME//workspace/sudoku/src/org/objectweb/easybeans/demo/sudoku/web/client/Sudoku.java
Created file $HOME/workspace/sudoku/Sudoku.launch
Created file $HOME/workspace/sudoku/Sudoku-shell
Created file $HOME/workspace/sudoku/Sudoku-compile
```

1.3. Calling EJB3 beans with an RPC service

1.3.1. Defining the interface of the service

The two interface of the service need to be present in the client package. In this howto, the package name is org.objectweb.easybeans.demo.sudoku.web.client.service;

Two interfaces have to be defined :

- The first one is the interface that will be implemented by the remote service.
- The other interface is the asynchronous interface. It is always a void method, a new parameter needs to be added on the method and the name of the interface is suffixed by `Async`.

1.3.1.1. Service interface

Here is the example of a service that takes an argument and return an object (A solver).

```
package org.objectweb.easybeans.demo.sudoku.web.client.service;

import org.objectweb.easybeans.demo.sudoku.web.client.api.GridData;
import org.objectweb.easybeans.demo.sudoku.web.client.api.ServiceException;

import com.google.gwt.user.client.rpc.RemoteService;

/**
* Interface used to call the servlet facade and then EJB3.
* @author Florent Benoit
*/
public interface ServletFacadeService extends RemoteService {

/**
* Solve a sudoku grid and send in return the solved grid.
* @param gridData the grid to solve
* @return the solved grid.
* @throws ServiceException if solving fails.
*/
GridData solve(GridData gridData) throws ServiceException;
}
```

Note

GridData needs to be a serializable object. But the serialization is not the JDK serialization. This class needs to implements the `com.google.gwt.user.client.rpc.IsSerializable` interface.

The Exception (ServiceException) is also implementing the `IsSerializable` interface.

1.3.1.2. Asynchronous interface

The signature is almost the same except the following :

- The return type is void.

- An extra argument is added in the method signature : the `AsyncCallback` object.
- The name of the class is ending with `Async` keyword.

```
package org.objectweb.easybeans.demo.sudoku.web.client.service;

import org.objectweb.easybeans.demo.sudoku.web.client.api.GridData;

import com.google.gwt.user.client.rpc.AsyncCallback;

/**
* Asynchronous interface. Same parameters of interface but all methods have an
* AsyncCallback parameter and are void !.
* @author Florent Benoit
*/
public interface ServletFacadeServiceAsync {

/**
* Asynchronous call to the solve method.
* @param gridData the grid to solve
* @param callback the callback to use for this method
*/
void solve(GridData gridData, AsyncCallback callback);
}
```

Note

The ServiceException is not thrown by the solve() method of Async interface. The exceptions will be reported into the `onFailure(Throwable t)` method of the callback.

1.3.2. Classes used by the interfaces

1.3.2.1. The GridData class

This class implements `IsSerializable` interface.

```
package org.objectweb.easybeans.demo.sudoku.web.client.api;

import com.google.gwt.user.client.rpc.IsSerializable;

/**
* Represents the data of a sudoku grid.
* It is a serializable object (gwt) used by remote service.
* @author Florent Benoit
*/
public class GridData implements IsSerializable {
...
}
```

1.3.2.2. The ServiceException exception

This class implements `IsSerializable` interface and it stores the message.

Also an empty constructor is required for the serialization.

```
package org.objectweb.easybeans.demo.sudoku.web.client.api;

import com.google.gwt.user.client.rpc.IsSerializable;

/**
* Exception thrown by the remote service. <br />
* Exception needs to be serializable (gwt)
* @author Florent Benoit
*/
public class ServiceException extends Exception implements IsSerializable {

/**
* The message of the exception.
*/
private String msg;


/**
```

```
* Empty message.
*/
public ServiceException() {
super();
}

/**
* Builds an exception with a given message.
* @param msg the message of the exception.
*/
public ServiceException(final String msg) {
super(msg);
this.msg = msg;
}


/**
* Gets the message of the exception.
* @return the message of the exception.
*/
public String getMessage() {
return this.msg;
}

}
```

1.3.3. Implementation of the service

1.3.3.1. Implementation service class

The implementation of the service will run on the server side. Then it doesn't need to be present in the client package (No JavaScript transformation). The class will be in the server package.

The service is provided as a servlet. Then, the class needs to extend the `com.google.gwt.user.server.rpc.RemoteServiceServlet` class.

Class will looks like :

```
package org.objectweb.easybeans.demo.sudoku.web.server.service;

import org.objectweb.easybeans.demo.sudoku.web.client.api.GridData;
import org.objectweb.easybeans.demo.sudoku.web.client.api.ServiceException;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
* Implementation of the service that runs on the server side. <br />
* All is delegate to the EJB3 session facade.
* @author Florent Benoit
*/
public class ServletFacadeServiceImpl extends RemoteServiceServlet implements
ServletFacadeService {


/**
* Solve a sudoku grid and send in return the solved grid.
* @param gridData the grid to solve
* @return the solved grid.
* @throws ServiceException if solving fails.
*/
public GridData solve(final GridData gridData) throws ServiceException {
...
}

}
```

1.3.3.2. Accessing to an EJB3 from this Remote Service.

A sessionBean facade will be used for delegating all the requests.

The service will use a getFacade() method. InitialContext is built then the facade is searched and the session bean is returned. The bean could be cached to avoid to get a new bean each time.

Here is the code of this method :

```
/**
* Gets the session facade bean.
* @return the session bean.
* @throws Exception if facade is not retrieved
*/
private SudokuFacade getFacade() throws Exception {
SudokuFacade sudokuFacade = null;

Context initialContext = null;
Hashtable<String, String> env = new Hashtable<String, String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.objectweb.carol.jndi.spi.MultiOrbInitialContextFactory");
initialContext = new InitialContext(env);
sudokuFacade = (SudokuFacade) initialContext.lookup("SudokuFacade");
return sudokuFacade;
}
```

Note

The JNDI name is very simple as the attribute mappedName was used on the @Stateless annotation of the EJB3. No Provider URL is used as the service and the EJB3s are running with the same registry. Default URL will be ok.

1.3.3.3. Calling facade bean from the remote service

When an operation needs to be done on the facade session bean, it is done by using the following way :

```
try {
... = getFacade().methodName();
} catch (Exception e) {
throw new ServiceException(e.getMessage());
}
```

The exception thrown are wrapped in ServiceException. Only the message is kept and thrown to the client. Error could be printed in the error log to have the full trace on the server side.

1.3.4. Calling Remote service from client side.

1.3.4.1. The gwt.xml file

The service needs to be added in an xml file in order to declare it. In this howto, the file is named `Sudoku.gwt.xml`.

The servet element is added in this file with the class implementing the service and the endpoint of this service (/facade is the following example).

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

<!-- Specify the app entry point class. -->
<entry-point class='org.objectweb.easybeans.demo.sudoku.web.client.Sudoku'/>

<servlet path='/facade'
class='org.objectweb.easybeans.demo.sudoku.web.server.service.ServletFacadeServiceImpl'/>

</module>
```

1.3.4.2. Calling service from client side.

1.3.4.2.1. Getting the service on client side

An endpoint needs to be created. The path of the endpoint will use the name of the module and then /facade (the same entry than in the gwt.xml file)

Endpoint will be something like "http://localhost:8080/org.objectweb.easybeans.demo/facade".

By using GWT.getModuleBaseURL(), the port and the host don't need to be known.

```
ServletFacadeServiceAsync servletFacadeServiceAsync = (ServletFacadeServiceAsync) GWT
.create(ServletFacadeService.class);
ServiceDefTarget endpoint = (ServiceDefTarget) servletFacadeServiceAsync;
endpoint.setServiceEntryPoint(GWT.getModuleBaseURL() + "/facade");
```

1.3.4.2.2. Creating the callback

Before invoking the service, a callback needs to be built. The interface of the service that is used is the asynchronous interface. As this interface requires a callback, it needs to be constructed before.

A callback provides two methods :

- An `onSuccess(final Object result)` method if the call completes with success.
- An `onFailure(final Throwable caught)` method if call fails.

The result object in the case of the resolve method can be casted in GridData as it is the return type of the original method's interface.

```
final AsyncCallback solvedCallback = new AsyncCallback() {
/**
* Called when an asynchronous call completes successfully. It is
* always safe to downcast the parameter (of type
* <code>Object</code>) to the return type of the original method
* for which this is a callback.
*/
public void onSuccess(final Object result) {
GridData solvedGridData = (GridData) result;
...
}

/**
* Called when an asynchronous call fails to complete normally.
* @param caught the failure.
*/
public void onFailure(final Throwable caught) {
...
}
};
```

1.3.4.2.3. Invoking the service

The invocation on the service is done by providing both arguments of the original method and the callback on the asynchronous interface.

```
serviceAsycn.solve(gridData, solvedCallback);
```

1.4. Packaging of the application for running on a web container.

1.4.1. Creating the web.xml file

A `web.xml` file has to be created to defines a servlet for the remote service that was defined and the mapping. The mapping has to be the same that was used in the client when searching the endpoint to use.

Here is an example of the xml file :

```
<?xml version="1.0" encoding="ISO-8859-1">
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.4">

<servlet>
<servlet-name>ServletFacadeService</servlet-name>

<servlet-class>org.objectweb.easybeans.demo.sudoku.web.server.service.ServletFacadeServiceImpl</
servlet-class>
```

```
</servlet>

<servlet-mapping>
<servlet-name>ServletFacadeService</servlet-name>
<url-pattern>/facade</url-pattern>
</servlet-mapping>

</web-app>
```

1.4.2. Generating the AJAX client that will be embedded into the war file

GWT provides some shell script to generate the client's side. It's better to use ant task.

First, a classpath needs to be defined. This classpath should include GWT libraries and the path to the source code of the Entry point :

```
<path id="gwt.classpath">
<pathelement location="${project.dir}/src" />
<pathelement location="${gwt.sdk.location}/gwt-user.jar" />
<pathelement location="${gwt.sdk.location}/gwt-dev-windows.jar" />
</path>
```

Then, code can be compiled :

```
<target name="compile" description="Compile demo" depends="init">
<javac srcdir="${src.dir}" destdir="${classes.dir}" debug="on">
<classpath refid="base.classpath" />
</javac>
</target>
```

And at the end, GWT generation can be done :

```
<target name="generate-gwt" depends="compile">
<java classname="com.google.gwt.dev.GWTCompiler" fork="true">
<arg value="-out" />
<arg value="${dist.www.dir}" />
<arg value="org.objectweb.easybeans.demo.sudoku.web.Sudoku" />
<classpath refid="base.classpath" />
</java>
</target>
```

1.4.3. Packaging of the .war file

The AJAX client will be bundled in a war file and it will contains the GWT runtime library.

The war task of ant can be used to do the package :

The output generated by the GWT compiler is added at the root of the WAR file, the libraries go in the `WEB-INF/lib` folder while all classes go in `WEB-INF/classes` folder.

```
<target name="build.war"
description="Build war file"
depends="compile, generate-gwt, removejavax">
<war jarfile="${dist.webapps.dir}/sudoku.war"
webxml="${resources.dir}/web.xml">
<fileset dir="${dist.www.dir}/org.objectweb.easybeans.demo.sudoku.web.Sudoku" />
<lib dir="${tmp.dir}">
<include name="gwt-user.jar" />
</lib>
<classes dir="${classes.dir}">
<include name="**/**" />
</classes>
</war>
</target>
```

Warning

The gwt-user.jar provided by Google contains javax.servlet.* classes and then it won't be deployed on Tomcat container. These classes have to be removed. An ant target is doing this job :

```
<target name="removejavax" depends="init">
<mkdir dir="${tmp.dir}/classes" />
<unjar src="${lib.dir}/gwt-user.jar" dest="${tmp.dir}/classes">
<patternset>
<exclude name="javax/**" />
</patternset>
</unjar>
<jar jarfile="${tmp.dir}/gwt-user.jar">
<fileset dir="${tmp.dir}/classes" />
</jar>
</target>
```

1.5. Deploying applications for GWT/EasyBeans

The war file generated by Ant should be copied into `CATALINA_HOME/webapps` folder while the EJB3 implementation should be copied into `CATALINA_HOME/ejb3s` folder.

Documentation on how to start Tomcat/EasyBeans can be found in the documentation section [http://wiki.easybeans.org/xwiki/bin/Main/Documentation] of EasyBeans web site.

1.6. Launch of the Google Web Toolkit tools

The GWT toolkit create a .launch script allowing to launch an embedded browser and an embedded web container allowing to debug applications within Eclipse. This script has to be modified for EasyBeans access.

The following lines have been added :

```
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ow_carol.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ow_carol_cmi.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ow_easybeans_api.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ow_easybeans_core.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ejb-2_1-api.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ow_ejb3_core_api.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
<listEntry value="&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;&#13;&#10;&lt;runtimeClasspathEntry
externalArchive=&quot;/easybeans-sudoku/lib/ow_ejb3_persistence_api.jar&quot; path=&quot;3&quot;
type=&quot;2&quot;/&gt;&#13;&#10;"/>
```

These libraries are used to access to the EasyBeans services.

1.7. Comments

This HowTo can be completed if some comments or questions are not answered by this guide. The idea was to illustrate how to call remote EJB3s running on EasyBeans by using Google Web Toolkit.

For developping EJB3 components, the user guide [http://wiki.easybeans.org/xwiki/bin/Main/Documentation]of EasyBeans can be used.