zenika
ARCHITECTURE INFORMATIQUE

Using GWT Generators for databinding

An introduction to GWT Generators

Zenika (www.zenika.com)
08/01/2007
by Jean-Philippe Dournel (jean-philippe.dournel@zenika.fr)

Preamble

The purpose of this tutorial is to explain how Deferred Binding works in GWT and the use of the Generator class to create classes dynamically. The example we will use thoughout this tutorial is the creation of an utility class which provides access to the properties of an object using the field names. This mechanism will prove very useful for solving recurrent problems like connecting a model to a GUI (also know as databinding).

Deferred Binding

In Java, reflection is a very powerful mechanism. Suppose we have two different classes, Employee and Visitor, which are subclasses of Person and we want the object *person to* indicate an instance of Employee or Visitor according to the context.

In Java we would use reflection like this:

```
// status contains "Employee" or "Visitor"
String status = getStatus();
Person person = (Person) Class.forName(status).newInstance();
```

Thus, the instance held by the object *person* is dynamically selected according to the character string *status*.

For several reasons, reflection isn't possible with GWT. The first one deal with code optimization: only methods and classes used in the application are translated into Javascript during compilation. The other reason is that it is not possible to load a class dynamically in Javascript.

To compensate this lack, GWT has Deferred Binding which consists in running the insertion of the required class during the compilation instead of the runtime. The creation of an instance is done thanks to the operation GWT.create (MyClass.class).

It is however not possible to call a class using character strings (such as "MyClass") with GWT, unlike Java's reflection. The name of the class must be described in a literal manner (MyClass.class). Now let's explain the difference between GWT.create (Classe.class) and new Class().

The difference lies in the compiling process. The new Class() call is directly translated into Javascript, whereas the GWT.create() call passes through the Generator which specifies which class must be instantiated.

zenika
ARCHITECTURE INFORMATIQUE

Presentation of the example

To explain how to use the Generator class, let us see the following example: we have a Person business class and we would like to be able to access its properties using their names.

In Java, we would use reflection with the following code:

```
private String getAttribute (Object object, String property) {
      Class objectClass = objet.getClass();
      // We suppose that to the property “property” corresponds the getter
      // getProperty()
      String getterName = "get" + property.substring(0, 1).toUpperCase()
                                + property.substring(1);
      Method method = objectClass.getMethod(getterName, null);
      String result = (String) methode.invoke(object, null);
      return result;
}
```

With GWT, we cannot use reflection. To implement the method getAttribute, we have to dynamically create a class which will be in charge of wrapping the business object (called by GWT.create ()) and which will implement the Wrapper interface.

```
public interface Wrapper {
      // the content object represents the object wrapped by the wrapper
      public void setContent(content Object);
      Object public getContent();
      public String getAttribute(String attr);
      public void setAttribute(String attr, String property);
}
```

Here is the class we want to wrap:

```
public class Person {

        private String name;
        private String firstName;

        public Person() {

        }

        public String getName () {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName (String firstName) {
                this.prenom = firstName;
        }

}
```

And here is the class we want to obtain by calling GWT.create (Person.class):

```
public class PersonWrapper implements Wrapper {

        private Personne content;

        public void setContent(Object content) {
                this.content = (Person) content;
        }

        public Object getContent () {
                return this.content;
        }

        public String getAttribute (String attr) {
                if (attr.equals ("name")) {
                        return content.getName();
                }
                if (attr.equals ("firstName")) {
                        return content.getFirstName();
                }
                return null;
        }

        public void setAttribute(String attr, String property) {
                if (attr.equals ("name")) {
                        this.content.setName(property);
                } else if (attr.equals("firstName")) {
                        this.content.setFirstName(property);
                }
        }
```

Use a Custom Generator

When the method GWT.create (MyClass.class) is called, GWT uses the default generator which returns the name of the class to instantiate (MyClass by default). To make GWT use another generator, it is mandatory to add the following lines in the module XML configuration file.

```
<generate-with class="com.zenika.tutorial.rebind.WrapperGenerator" >
       <when-type-assignable class=”
       com.zenika.tutorial.gwt.client.BusinessObject" />
</generate-with>
```

This piece of code makes GWT use the WrapperGenerator class for any call to an object which implements the BusinessObject interface. The BusinessObject interface is empty and is only used to identify classes which have to pass through the WrapperGenerator. The technique is slightly intrusive.

The WrapperGenerator class extends the abstract class Generator of GWT (contained in gwt-dev-'OS'.jar) and is located by convention in a package .rebind apart from the GWT module. This is normal since this class won't be compiled and won't provide a RPC service.

```
package com.zenika.tutorial.rebind;

import com.google.gwt.core.ext. Generator;
import com.google.gwt.core.ext.GeneratorContext;
import com.google.gwt.core.ext.TreeLogger;
import com.google.gwt.core.ext.UnableToCompleteException;

/**
* Method called at the time of the sequence of Deferred Binding initiated by
* the GWT.create call ().
* Return the name of the class to be intanciated.
*/
public class WrapperGenerator extends Generator {
       public String generate(TreeLogger logger, GeneratorContext context,
                     String typeClass) throws UnableToCompleteException {
              WrapperCreator binder = new WrapperCreator (logger, context,
                            typeClass);
              String className = binder.createWrapper();
              return className;
       }
}
```

The *generate* method requires different parameters:

- TreeLogger which is used to log messages
- GeneratorContext which manages metadata
- typeClass which is the name of the class passed as a parameter of GWT.create()

It returns the name of the class to instantiate. Here, we want it to return PersonWrapper instead of Person. This PersonWrapper class does not exist yet and will be created dynamically based on the class Person by the WrapperCreator class.

Generate a Class dynamically

The last stage is the creation of the wrapper WrapperCreator. We will describe this operation in details.

The parameters passed to the method 'generate' are retained as fields (done inside the constructor). The TypeOracle object extracted from GeneratorContext is a GWT class which provides access data (class name, methods, method parameters, etc) of the class used as parameter of the GWT.create method () (here the class Person), in a very similar fashion to Java's reflection.

```
public WrapperCreator (TreeLogger logger, GeneratorContext context,
            String typeName) {
      this.logger = logger;
      this.context = context;
      this.typeName = typeName;
      this.typeOracle = context.getTypeOracle();
}
```

Then the method createWrapper is called by WrapperGenerator. Using the typeOracle, it retrieves the name of the class required and calls the method getSourceWriter ().

```
public String createWrapper () {
      try {
            JClassType classType = typeOracle.getType(typeName);
            SourceWriter source = getSourceWriter(classType);
            …
      }
      …
}
public SourceWriter getSourceWriter (JClassType classType) {

      String packageName = classType.getPackage().getName ();
      String simpleName = classType.getSimpleSourceName() + "Wrapper";
      ClassSourceFileComposerFactory composer =
            new ClassSourceFileComposerFactory(packageName, simpleName);
      composer
      .addImplementedInterface("com.zenika.tutorial.gwt.client.Wrapper");
      PrintWriter printWriter = context.tryCreate (logger, packageName,
                  simpleName);
      if(printWriter == null) {
            return null;
      } else {
            SourceWriter sw = composer.createSourceWriter(context,
                  printWriter);
            return sw;
      }
}
```

SourceWriter manages the textual flow of the class to be generated. It has a similar behavior to FileWriter except that it adds indent management. The SourceWriter returned contains the package declarations, the class name and the implemented interfaces. The only thing left is the actual class body. For that purpose, we use the oracle to generate the methods getAttribute and setAttribute according to the properties of the Person object. Here is the snippet which creates the method getAttribute (the complete code of the class is also available, see downloads).

```
JMethod[] methods = classType.getMethods();
for (int i = 0; i < methods ; i++ ) {
      String methodName = methods[i].getName();
      JParameter[] methodParameters = methods [i].getParameters ();
      JType returnType = methods [i].getReturnType();
      if (methodName.startsWith("get") & methodParameters.length == 0) {
            source.println("if (attr.equals (\""
                        + methodName.substring(3).toLowerCase ()
                        + "\")) {" );
            source.indent();
            source.println("return content." + methodName + "();");
            source.outdent();
            source.print("} else");
      }
}
source.println("{");
source.indent();
source.println("return null;");
source.outdent();
source.println("}");
```

Use the wrapper

Now, to use the wrapper you only have to do this:

```
Wrapper wrapper = (Wrapper) GWT.create(Personne.class);
wrapper.setContent(person);
```

And you can access the data contained in person with the function getAttribute of the wrapper (for example, getAttribute (”name”)).

Here is a more concrete use example :

```
// Creation of the model, a simple pojo normally recovered a RPC service
final Person person = new Person("Robert", "Charlebois");
// Creation of the wrapper which allow data binding
Wrapper wrapper = (Wrapper) GWT.create(Person.class);
// wrap the model
wrapper.setContent(person);
// bind the name and the first name with two textboxes
RootPanel.get().add(Binder.bind(new TextBox (), wrapper, "firstName"));
RootPanel.get().add (Binder.bind(new TextBox (), wrapper, "name"));
Button button = new Button ("Inspect the new values of the model");
button.addClickListener (new ClickListener () {
        public void onClick(Widget sender) {
                // test to see if the IHM reflect well the new model values
                Window.alert(personne.getFirstName()+" "+personne.getName());
        }
});
RootPanel.get().add(button);
```

The End…

We provided a zip file containing the source code of the wrapper generator and a databinding example. You can download it here: GWT_Binding

I particularly thank Ray Cromwell who was one of the first people to document Generator on his blog http://timepedia.blogspot.com/.

This tutorial is an introduction to a more complete tutorial centered on databinding. We will see in particular how to create a Bind class which will deal with the creation of the wrapper in a completely generic way and the dynamic creation of Validators.