DWR in a Jiffy (quickstart tutorial)

by Benny Bottema - Saturday, January 19, 2008

http://www.bennybottema.com/2008/01/19/dwr-in-a-jiffy/

Direct Web Remoting**, or DWR in short**, is a technology that helps you ease the development of javascript based applications using a Java server. What it does is simply sit in between your javascript and your java classes and acts as marshaller for method calls from javascript to Java. In a nutshell: DWR provides Remote Procedure Calls for javascript and is a great library for use in Ajax enabled webapplications.

This isn't yet another blogpost about what DWR exactly is or why it is so cool; the objective of my post today is to simply set up a quick example of how to work with it. A quick tutorial if you will. I'm the kind of guy that likes many examples and diagrams and whatnot to take apart and learn from, so here's my contribution to the 'code by example' paradigm. If you want to know more about DWR before actually trying to use it, I suggest you check out the DWR homepage.

Contents

Setting the stage

One of the cool things (two sentences and I already contradict myself :), is that it is so easy to setup, as you will soon see. To get DWR to expose the classes to javascript, you first need to add the dwr library and define the specific classes. Basically, this is a 3-step dance:

1. Drop the dwr.jar you've downloaded from directwebremoting.org in the lib folder of your webapplication (ie. *dwr.jar*).
2. Add DWR to your web.xml as a servlet:

```
<servlet>
 <servlet-name>dwr-invoker</servlet-name>
 <display-name>DWR Servlet</display-name>
 <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
 <init-param>
```

```
    <param-name>debug</param-name>
    <param-value>true</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
   <servlet-name>dwr-invoker</servlet-name>
   <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>
```

3. Create a *dwr.xml*, which informs DWR about the classes you need exposed:

```
<dwr>
 <allow>
  <create creator="new" javascript="MyService">
   <param name="class" value="org.foo.MyServiceClass"/>
  </create>
 </allow>
</dwr>
```

Testing DWR

To check if you've done everything right, you can open up the DWR testpage *http://localhost/dwr/* on which all exposed classes and methods should be visible. Remember, this works because you configured a DWR servlet that triggers on */dwr/**. If you click on any of the classes that are exposed, you'll get a list of methods that are available to you javascript and are ready for testing right away. Note that this is all javascript already!

Specifying nothing but */dwr/* results in this testpage. But If you want to use a specific java class in your own page to call methods on, you'd call */dwr/TheService*. What happens is that the DWR library dynamically generates all the required javascript to communicate with the server through this servlet, runtime. Pretty cool huh?

Using DWR on the client side

To use the methods of the classes that are exposed through dwr.xml, you need to include a virtual javascript file with the name of such an exposed class. Since the classes are identified with the *javascript* attribute in dwr.xml, you'll use that identifier when importing the javascript file. In addition to the scriptfile for your javaclass, you need at least *engine.js* as well.

```
<script src="/dwr/interface/MyService.js" mce_src="/dwr/interface/MySe
rvice.js"></script>
<script src="/dwr/engine.js" mce_src="/dwr/engine.js"></script>
```

There are some other useful scripts you can include from DWR, such as *util.js*, which has a couple of nifty functions like $() (a shorthand version for document.getElementById()).

About DWR callbacks

To actually make use of the Java serviceclass, you simply call a method on the Myservice.js file. The catch here is that you need to provide a callback function as well. This is because of the fact that javascript runs asynchronously in the client and doesn't wait for Java to come back with an answer. Instead, DWR relies on the callback method you defined that will handle the result in due time. Note that the function you call on the serviceclass.js has all the parameters as defined in the Java class, safe for the callback parameter and peripheral parameters such as HttpSession or HttpRequest.

The callback function can only have one parameter, *data*, which bares the result of the DWR servicecall.

Take a look:

```
// call function on DWR generated service.js
MyService.getInfo(arg1, arg2, parseResult);

// callback function that parses whatever result comes back
function parseResult(data) {
  // do stuff with result data
}
```

On a side note, there are a couple of varying ways of calling a function on a serviceclass. For example, you can have the callback function as first or last argument, or not at all. You can also specify a time-out and some other options. For more details on check out the DWR page on this usage subject.

Matching arguments types

DWR already provides most common conversions such as Strings, Dates, Arrays (nested as well) and a host of other types, including stuff like Vector. If you'd like to use your own type for a parameter, you will need to create a javascript version with the same fields. Here's an example:

```
// Java version:
public Person {
 private String name;
 private int age;
 private Date[] appointments;

 // getters and setters ...
```

```
}

// javascript version:
var person = {
 name:"Fred Bloggs",
 age:42,
 appointments:[ new Date(), new Date("1 Jan 2008") ]
};

MyService.setPerson(person);
```

To make these conversions possible, you need to inform DWR about these ‘custom’ classes. These so called ‘beans’ allow DWR converters to marshal these classes from/to javascript.

More on DWR converters here:

- /converters
- /converters/bean
- /converters/collection
