



# Introduction to DWR (Direct Web Remoting)

Sang Shin  
Java Technology Architect  
Sun Microsystems, Inc.  
[sang.shin@sun.com](mailto:sang.shin@sun.com)  
[www.javapassion.com](http://www.javapassion.com)

Welcome to the introduction to DWR (Direct Web Remoting) presentation. My name is Sang Shin from Sun Microsystems.

## Disclaimer & Acknowledgments

- Even though Sang Shin is a full-time employee of Sun Microsystems, the contents here are created as his own personal endeavor and thus does not reflect any official stance of Sun Microsystems
- Many slides are borrowed from DWRIntro presentation authored by Joe Walker (with his permission)
- Most slides are created from contents from DWR website <http://getahead.ltd.uk/dwr/>

## Topics

- What is and Why DWR?
- Steps for building DWR-based AJAX application
- Registering callback functions
- Utility functions
- Engine functions
- Handling errors and warnings
- Security
- DWR and Web application frameworks

3

This is the list of the topics we will be talking about in this presentation.

First, I will talk about what DWR is and some of the motivations of using DWR for building AJAX applications. Then I will go over steps you want to follow to build a simple DWR-based AJAX application.

Then I will talk about how to register callback functions that can be called asynchronously. I will then talk about some of the utility and engine functions. These are JavaScript libraries.

I will then cover how error handling is done. I will address security aspects of DWR and finally touch upon how DWR can be used with other frameworks such as JSF.

# What is DWR?

First, let's talk about what DWR is.

## What is DWR?

- Is a Java and JavaScript open source library which allows you to write Ajax web applications
  - > Hides low-level *XMLHttpRequest* handling
- Specifically designed with Java technology in mind
  - > "Easy AJAX for Java"
- Allows JavaScript code in a browser to use Java methods running on a web server just as if they were in the browser
  - > Why it is called "Direct remoting"

5

Direct Web Remoting is Java and JavaScript open source library which allows you to write AJAX web application.

What is unique about DWR, however, is that it is specifically designed with Java technology in mind. And that is the reason why some people call DWR as "Easy AJAX for Java".

Now what do I mean by that statement? DWR allows JavaScript code in a browser to invoke server side Java methods just as if they were in the browser. That is why DWR is called direct remoting.

I will explain this aspect further by using example code soon so if you don't understand what I mean by this, don't worry about it.

## Why DWR?

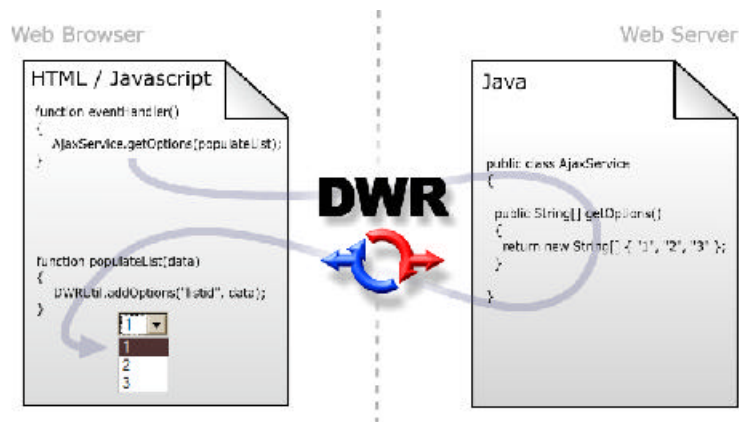
- Without DWR, you would have to create many Web application endpoints (servlets) that need to be address'able via URI's
- What happens if you have several methods in a class on the server that you want to invoke from the browser?
  - > Each of these methods need to be addressable via URI whether you are using *XMLHttpRequest* directory or client-side only toolkit such as Dojo or Prototype
  - > You would have to map parameters and return values to HTML input form parameters and responses yourself
- DWR comes with some JavaScript utility functions 6

What is the motivation of using DWR? Without using DWR, you would have to create many web application endpoints (servlets) that need to be addressable via URI's. And in many cases, this is not practical.

For example, what would you do if you have several methods in a class on the server that you want to invoke from a browser. First of all, each of this method needs to be address'able via URI if you are using *XMLHttpRequest* directly or even if you are using some client-side only toolkit such as Dojo or Prototype. Even if you solve this problem through some magic, you still need to map parameters and return values to HTML inout form parameters and responses yourself. DWR handles these issue for you.

DWR also comes with some JavaScript utility libraries.

## How DWR Works



7

So this is how DWR works. Let's say you have a Java class in the backend called `AjaxService`. It has a method called `getOptions()`.

On the client, you can write your JavaScript code in which you invoke the `getOptions()` method of the `AjaxService` class using the very RMI like syntax.

```
AjaxService.getOptions(populateList)
```

Now only one difference is that the `getOptions()` method on the server does not have any parameter while the one in the client side has a parameter, in this case, `populateList`.

This additional parameter is a callback function that will be invoked asynchronously. We will talk about this in a bit more detail later in this presentation.

So please note that, except the callback function as an additional parameter, the calling of a method of a remote Java class looks very much like RMI syntax which most Java developers are very much familiar with.

## DWR Consists of Two Main Parts

- A DWR-runtime-provided Java Servlet running on the server that processes incoming DWR requests and sends responses back to the browser
  - > [uk.ltd.getahead.dwr.DWRServlet](http://uk.ltd.getahead.dwr.DWRServlet)
  - > This servlet delegates the call to the backend class you specify in the [dwr.xml](#) configuration file
- JavaScript running in the browser that sends requests and can dynamically update the webpage
  - > DWR handles *XMLHttpRequest* handling

source: <http://getahead.ltd.uk/dwr> 8

So DWR framework consists of two main parts - one on the server and the other on the client.

On the server side, the DWR runtime includes a Java Servlet that receives the incoming DWR requests and dispatch them to the Java class. The name of the servlet is `DWRServlet`. And the Java class you want to delegate the requests to are specified in the `dwr.xml` configuration file.

On the client side, the DWR hides creation of `XMLHttpRequest` object and handling of it.

## How Does DWR Work?

- DWR dynamically generates a matching client-side Javascript class from a backend Java class
  - > Allows you then to write JavaScript code that looks like conventional RPC/RMI like code, which is much more intuitive than writing raw JavaScript code
- The generated JavaScript class handles remoting details between the browser and the backend server
  - > Handles asynchronous communication via *XMLHttpRequest* - Invokes the callback function in the JavaScript
  - > You provide the callback function as additional parameter
  - > DWR converts all the parameters and return values between client side Javascript and backend Java

9

So how does DWR work? DWR dynamically generates a matching client-side JavaScript class from a backend Java class. This is why DWR allows you to write JavaScript code that looks like conventional RMI like code, which is much more intuitive than writing raw JavaScript code.

The DWR-generated JavaScript class handles remoting details between the browser and backend server. By remoting details, I mean handling of asynchronous communication through the usage of *XMLHttpRequest* object and invocation of the callback function on the client side. The callback function is something you provide. The remoting details also includes conversion of all parameters and return values between client side and backend Java code.



# Steps for Building DWR-based AJAX Application

OK. Let's take a look at the step by step process of building DWR-based AJAX application.

## Steps to Follow

1. Copy `dwr.jar` file into the `WEB-INF/lib` directory of your web application
  - ? `dwr.jar` contains DWR runtime code including the DWR servlet
2. Edit `web.xml` in the `WEB-INF` directory
  - ? DWR servlet mapping needs to be specified
3. Create `dwr.xml` file in the `WEB-INF` directory
  - ? You specify which class and which methods of the backend service you want to expose
4. Write client-side JavaScript code, in which you invoke methods of remote Java class (or classes) in RPC/RMI-like syntax
5. Build, deploy, test the application

11

These are the steps you will take to develop DWR application. First of all, you will have to include DWR server side runtime jar file in your web application.

Second, you edit `web.xml` file to provide servlet mapping for the DWR servlet.

Next, you create `dwr.xml` file under `WEB-INF` directory. As we will see later on, the `dwr.xml` file is the DWR specific configuration file in which you specify which class and what methods you want to expose for remoting.

Next, you write JavaScript code in which you invoke methods of Java class in RMI like syntax.

Finally, you build and test the application.

So let's go over each of these steps in a bit more detail.

## Step #1: Copy *dwr.jar* File in the WEB-INF/lib Directory

- [dwr.jar](#) contains DWR runtime code including the DWR servlet
- You can get [dwr.jar](#) file from <http://getahead.ltd.uk/dwr/download>
- The latest version is 2.0 (as of June 2007)

12

Step #1, coping *dwr.jar* file into the WEB-INF/lib directory of your web application. As was mentioned before, *dwr.jar* file contains DWR runtime code including DWR servlet.

You can get the *dwr.jar* file from the *dwr* website mentioned in the slide.

## Step #2: Edit **web.xml** in the **WEB-INF** directory

```
<!-- Configure DWR for your Web application -->
<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>
</servlet>

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

13

The next step is to edit web.xml file so that you provide the servlet mappig for the DWR servlet. All DWR-based web application has to have this servlet mapping.

## Step #3: Create **dwr.xml** file in the WEB-INF directory

- The **dwr.xml** config file defines what classes and what methods of those classes DWR can create and remote for use by client-side Javascript code
- Suppose I have a Java class called **mypackage.Chat** and I want to create a matching JavaScript class called **Chat**
  - > **mypackage.Chat** Java class (server)
  - > **Chat** JavaScript class (client)

14

Next, you are going to create **dwr.xml** in the WEB-INF directory of your application.

As was mentioned before, the **dwr.xml** configuration file defines what classes and what methods of those classes you want to expose.

For example, let's say that I have a Java class called "Chat" under "mypackage" package and I want to create a matching JavaScript class called "Chat".

## Step #3: Create **dwr.xml** file in the WEB-INF directory

```
<!DOCTYPE dwr PUBLIC
"-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
"http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <create creator="new" javascript="Chat">
      <param name="class" value="mypackage.Chat"/>
    </create>
    <convert converter="bean" match="mypackage.Message"/>
  </allow>
</dwr>
```

15

This is how you specify them in the dwr.xml file. The name of the JavaScript class is specified as the value of “javascript” attribute of the <create>. And the name of the Java class is specified as the value of “value” attribute of <param>.

The <convert> element indicates to the DWR runtime that the properties of the JavaBean called Message need to be marshalled.

## Step #4a: Write Client-side JavaScript code in which you invoke methods of a Java class

```
<!-- You have to include these two JavaScript files from DWR -->
<script type='text/javascript' src='dwr/engine.js'></script>
<script type='text/javascript' src='dwr/util.js'></script>

<!-- This JavaScript file is generated specifically for your application -->
<script type='text/javascript' src='dwr/interface/Chat.js'></script>
```

16

The next step is to write client JavaScript code. Every DWR JavaScript code include DWR-provided JavaScript utility libraries.

It also needs to include application specific JavaScript library that is generated by the DWR framework.

## Step #4b: Write JavaScript client code in which you invoke methods of a Java class

```
<script type='text/javascript'>
  function sendMessage(){
    var text = DWRUtil.getValue("text");
    DWRUtil.setValue("text", "");

    // Invoke addMessage(text) method of the Chat class on
    // the server. The gotMessages is a callback function.
    // Note the RPC/RMI like syntax.
    Chat.addMessage(gotMessages, text);
  }

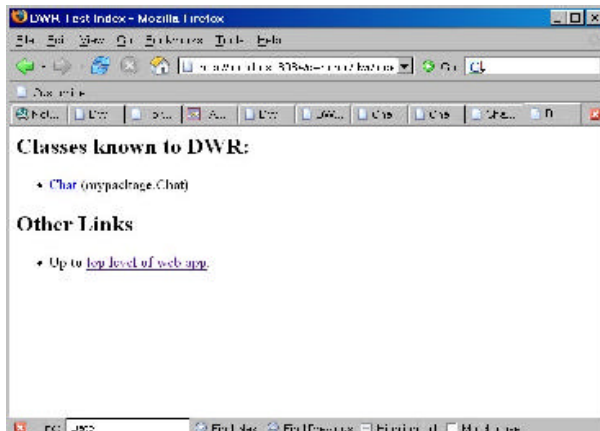
  function checkMessages(){
    // Invoke getMessages() method of the Chat class on
    // the server. The gotMessages is a callback function.
    Chat.getMessages(gotMessages);
  }
}
</script>
```

17

Now in this slide, you see examples of how methods of remote Java class are being invoked on the client side. Now, please note that there is an additional parameter called "gotMessages". This is the callback function.

## Step #5: Build, Deploy, & Test

- You can see the test page of your application
  - > `http://localhost:8084/<Your-Application-Context>/dwr`

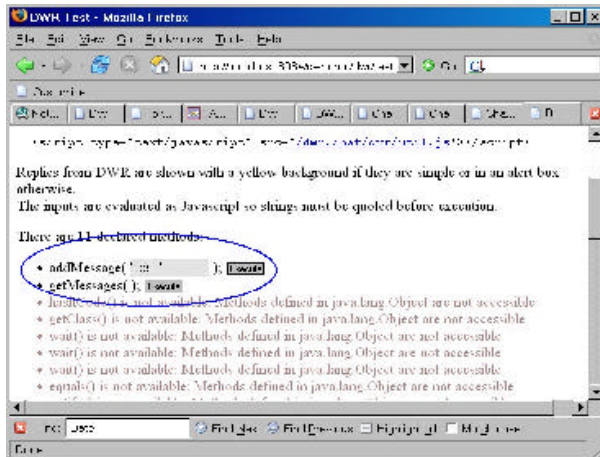


18

You can then build, deploy, and test your application. DWR provides useful testing feature. If you append “/dwr” at the end of your application context, it will display a screen in which you can perform the testing.

## Step #5: Build, Deploy, and Test

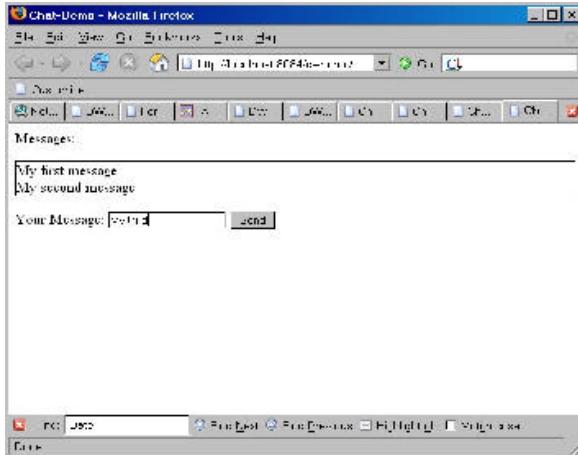
- You can actually test the interaction with the server



19

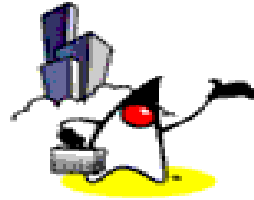
Through this test page, you can invoke all the methods that are exposed for remoting. You can even enter values as parameters.

## Step #6: Run the Application



20

Finally you run your application.



# Demo: Building & Running Chat Application using NetBeans IDE

21

So let me give you a demo on building and running Chat application using NetBeans IDE.



# Registering Callback Function for AJAX-based Asynchronous Invocation

We already talked about how you specify the callback function as an additional parameter to a method. Let's look into it in a bit more detail.

## How DWR Handles Asynchronous AJAX-Call

- Calling JavaScript function at the client needs to be done asynchronously while calling a Java method (at the server) is synchronous
  - > DWR handles this mismatch
- DWR provides a scheme for registering a callback function at the client
  - > You pass the callback function as an additional parameter
  - > The callback function is called when the data is returned from the server - this is AJAX behavior

23

Because calling JavaScript function at the client need to be done asynchronously while calling a Java method at the server is basically synchronously, the DWR has to handle this mismatch.

The DWR provides a scheme for registering a callback function as an additional parameter when a method is invoked. This callback function is called when the data is returned from the server asynchronously.

## Example 1: How Callback Function is Registered

- Suppose we have a Java method that looks like this:

```
// Server side Java code
public class MyJavaClass {
    public String getData(int index) { ... }
}
```

- We can use this from Javascript as follows:

```
// Callback function to be called
function handleGetData(str) {
    alert(str);
}
// The callback function is passed as an additional parameter
MyJavaScriptClass.getData(42, handleGetData);
```

24

Let's see an example. Here let's say we have a server side Java class called `MyRemoteJavaClass`, which has a method called `getData(int index)`. It has one parameter called `index`.

On the client side, there is a callback function called `handleGetData`. The way you pass this callback function is to pass it as an additional parameter either as the first parameter or the last parameter.

## Example 2: How Callback Function is Registered

- Suppose we have a Java method that looks like this:

```
// Server side Java code
public class MyJavaClass {
    public String getData(int index) { ... }
}
```

- Callback function can be in-lined

```
MyJavaScriptClass.getData(42,
    function(str) { alert(str); });
```

25

The callback function can be inlined as expected.

## Example 3: How Callback Function is Registered

- Suppose we have a Java method that looks like this:

```
// Server side Java code
public class MyRemoteJavaClass {
    public String getData(int index) { ... }
}
```

- You can use **Meta-data object**

```
MyRemoteJavaScriptClass.getData(42,
    {callback:function(str) { alert(str); }});
```

26

You can also use metadata object to pass the callback function.

## Example 4: How Callback Function is Registered

- Suppose we have a Java method that looks like this:

```
// Server side Java code
public class MyRemoteJavaClass {
    public String getData(int index) { ... }
}
```

- You can specify **timeout and error handler** as well

```
MyRemoteJavaScriptClass.getData(42,
    {callback:function(str) { alert(str); }}
    timeout:5000,
    errorHandler:function(message)
        { alert("Oops: " + message); }
);
```

27

You can also specify the timeout and error handler function if you are using meta-data object.

# Converters

Now let's talk about converters.

## Converters

- Converter marshals data between client and server
- Types of converters provided by DWR
  - > Basic converters
  - > Date converter
  - > Bean and Object converters
  - > Array converter
  - > Collection converter
  - > DOM Objects
- You can create your own converters
  - > Rarely needed

29

A converter marshals data between client and server. The DWR provides several converters including Data converter, Bean and Object converter, Array and collection converts.

You can create your own converters but it is rarely needed.

## Basic Converters

- Handles
  - > boolean, byte, short, int, long, float, double, char, java.lang.Boolean, java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.Character, java.math.BigInteger, java.math.BigDecimal and java.lang.String
- No need to have a <convert ...> element in the <allow> section in *dwr.xml* to use them
  - > They are enabled by default

30

The basic converts handles primitive types and corresponding Java classes. For these types, you do not need to specify <convert> element because they are enabled by default.

## Date Converter

- Marshalls between a Javascript *Date* and a *java.util.Date*, *java.sql.Date*, *java.sql.Timestamp* or *java.sql.Time*
- Is enabled by default
  - > Like Basic converters

The Date converter marshalls between a JavaScript Date class and Date class on the Java side.

## Bean and Object Converters

- These are not automatically enabled
  - > DWR makes sure that it has a permission before it touches any of your code
  - > You have to specify your instruction in the [dwr.xml](#)
- Bean converter will convert POJOs into JavaScript associative arrays and back again
- Object converter is similar except that it work on object members directly rather than through getters and setters

32

Bean and Object converters are not automatically enabled mainly because DWR makes sure that ti

## Example: Bean Converter

- Enable the bean converter for a single class  
`<convert converter="bean" match="your.full.package.BeanName"/>`
- Allow conversion of any class in the given package, or sub package  
`<convert converter="bean" match="your.full.package.*"/>`
- Allow conversion of all Java Beans  
`<convert converter="bean" match="*/>`

## Advanced Converters

- Declare new converters in the `<init>` element in *dwr.xml*
- Use `$` for inner classes
- `BeanConverter` can restrict exported properties

# Utility Functions

## Utility Functions in util.js

- DWR comes with *util.js*
- The *util.js* contains a number of utility functions to help you update your web pages with JavaScript data
- You can use it outside of DWR because it does not depend on the rest of DWR to function

## List of Utility Functions

- \$(id)
- getValue, getValues, setValue, setValues
- addRows and removeAllRows
- addOptions and removeAllOptions
- getText
- onReturn
- selectRange
- toDescriptiveString
- useLoadingMessage

## \$(id)

- \$(id) is the same thing as
  - > *document.getElementById(id)* in DOM API
  - > *dojo.byId(id)* in Dojo toolkit

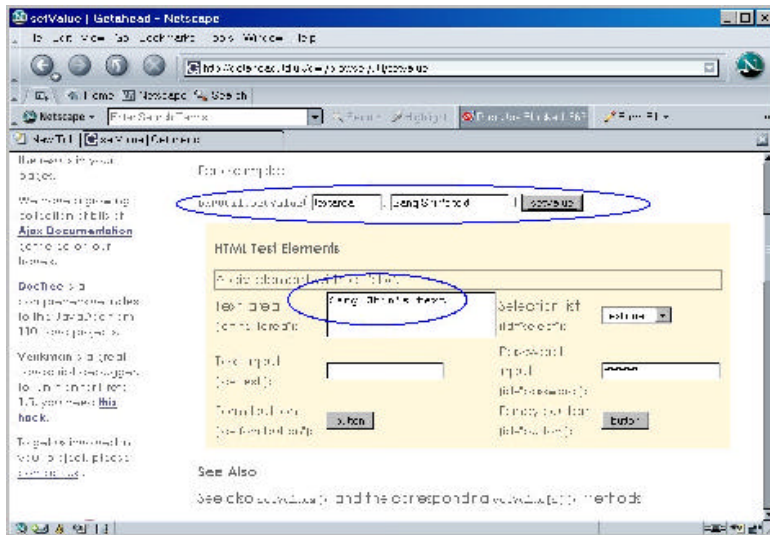
## getValue, getValues

- `DWRUtil.getValue(id);`
- This gets the value(s) out of the HTML elements without you needing to worry about how a selection list differs from a div
- This method works for most HTML elements including selects (where the option with a matching value and not text is selected), input elements (including textarea's) div's and span's

## setValue, setValues

- `DWRUtil.setValue(id, value);`
- This finds the element with the `id` specified in the first parameter and alters its contents to be the value in the second parameter.
- This method works for almost all HTML elements including selects (where the option with a matching value and not text is selected), input elements (including textarea's) div's and span's.

# setValue, setValues



## Manipulating Tables: addRows

- `DWRUtil.addRows(id, array, cellfuncs, [options]);`
  - > Adds rows to a table element specified by `id`
- Parameters
  - > `id`: The id of the table element (preferably a `tbody` element)
  - > `array`: Array (or object from DWR 1.1) containing one entry for each row in the updated table
  - > `cellfuncs`: An array of functions (one per column) for extracting cell data from the passed row data
  - > `options`: An object containing various options

## Manipulating Tables: `removeAllRows(id)`;

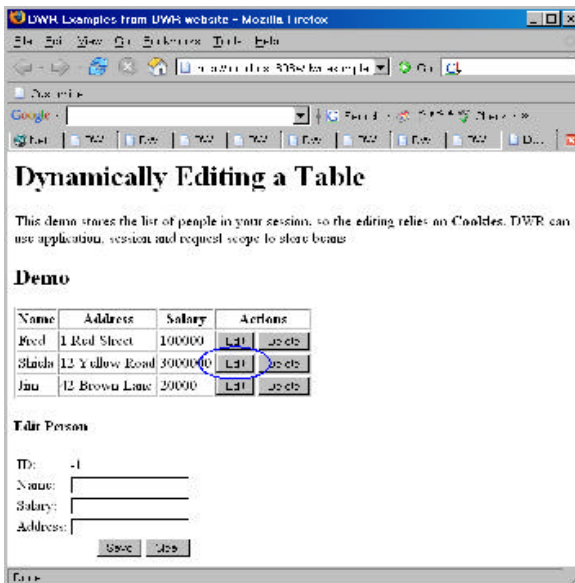
- `DWRUtil.removeAllRows(id)`;
  - > Removes all the rows in a table element specified by `id`
- Parameters
  - > `id`: The id of the table element (preferably a `tbody` element)

## Exemple #1 : Manipulating Tables

```
<script type="text/javascript">
  // Functions to be passed to DWRUtil.addRow
  var getName = function(person) { return person.name };
  var getDoB = function(person) { return person.address };
  var getSalary = function(person) { return person.salary };
  var getEdit = function(person) {
    return '<input type="button" value="Edit"
onlick="readPerson('+person.id+')"/>';
  };
  var getDelete = function(person) {
    return '<input type="button" value="Delete"
onlick="deletePerson('+person.id+', \''+person.name+'\')"/>';
  };

  // Callback function for getAllPeople method
  // The table is reconstructed
  function fillTable(people) {
    DWRUtil.removeAllRows("peoplebody");
    DWRUtil.addRow("peoplebody", people, [ getName, getDoB, getSalary,
getEdit, getDelete ])
  }
</script>
```

# Example #1: Manipulating Tables



Dynamically Editing a Table

This demo stores the list of people in your session, so the editing relies on Cookies. DWR can use application, session and request scope to store beans.

**Demo**

Name	Address	Salary	Actions
Kevin	1 Red Street	100000	<a href="#">Edit</a> <a href="#">Delete</a>
Blair	12 Yellow Road	300000	<a href="#">Edit</a> <a href="#">Delete</a>
Jan	12 Brown Lane	20000	<a href="#">Edit</a> <a href="#">Delete</a>

**Edit Person**

ID:

Name:

Salary:

Address:

## Example #2: Manipulating Tables

Unaltered	Altered	Button	Count
Africa	AFRICA	<input type="button" value="Test" onclick=alert("HI");//>	6
America	AMERICA	<input type="button" value="Test" onclick=alert("HI");//>	7
Asia	ASIA	<input type="button" value="Test" onclick=alert("HI");//>	8
Australasia	AUSTRALASIA	<input type="button" value="Test" onclick=alert("HI");//>	9
Europe	EUROPE	<input type="button" value="Test" onclick=alert("HI");//>	10

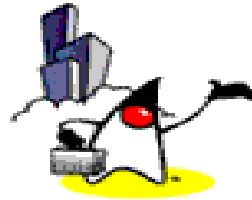
```

var callFuncs = [
  function(data) { return data; },
  function(data) { return data.toUpperCase(); },
  function(data) {
    return '<input type="button" value="Test" onclick=alert("HI");//>';
  },
  function(data) { return count44; }
];

var count = 1;
<%= ucil.addRow( "demo1", [{"Africa", America, Asia, Australasia, Europe} ] ); callFuncs:


<%= ucil.removeAllRows("demo1");


```



**Demo: Utility Functions  
from  
<http://getahead.ltd.uk/dwr/browser/util>**



# Engine Functions

## engine.js Functions

- *engine.js* is vital to DWR since it is used to marshal calls from the dynamically generated interface javascript function
- *engine.js* also contain set options methods
  - > Options may be set globally (using a DWREngine.setX() function) or at a call or batch level (using call level meta data e.g { timeout:500, callback:myFunc })
  - > A batch is several calls that are sent together.

## Engine Options

- Robustness
  - > errorHandler, warningHandler, timeout
- UI clues
  - > preHook, postHook
- Remoting options
  - > method, verb, async
- Call sequencing
  - > ordered, callback
- Future
  - > skipBatch, onBackButton, onForwardButton



# Handling Errors and Warnings

## Built-in Global Error Handlers

- Whenever there is some sort of failure, DWR calls an error or warning handler (depending on the severity of the error) and passes it the message
  - > This method could be used to display error messages in an alert box or to the status bar
- DWR provides built-in global error handlers
  - > *errorHandler* for errors
  - > *warningHandler* for warnings
- You can set the global error handlers with your own
  - > *DWREngine.setErrorHandler(youOwnErrorHandler);*
  - > *DWREngine.setWarningHandler(youOwnWarningHandler);*

## You Can Also Specify Handler In a Call

```
Remote.method(params, {  
  callback:function(data) { ... },  
  errorHandler:function(errorString, exception) { ... }  
});
```

## You Can Also Specify Handler In Batch Meta-data form

```
// Start the batch
DWREngine.beginBatch();

Remote.method(params, function(data) { ... });
// Other remote calls

DWREngine.endBatch({
  errorHandler:function(errorString, exception) { ... }
});
```



# Setting Global Timeout

## Setting Global Timeout

- *DWREngine.setTimeout()* function sets the timeout for all DWR calls
  - > A value of 0 (the default) turns timeouts off
  - > The units passed to *setTimeout()* are milli-seconds
  - > If a call timeout happens, the appropriate error handler is called
- You can set the timeout on an individual call level

```
Remote.method(params, {
  callback:function(data) { alert("it worked"); },
  errorHandler:function(message) { alert("it broke"); },
  timeout:1000
});
```



# How to Pass Servlet Objects as Parameters

## Handling Servlet Objects (Implicit Objects)

- If a Java method has, a servlet object as a parameter, ignore it in the matching JavaScript method DWR will fill it in
  - > HttpServletRequest
  - > HttpServletResponse
  - > HttpSession
  - > ServletContext
  - > ServletConfig

## Handling Servlet Objects (Implicit Objects)

- For example if you have remoted a class like this:

```
public class Remote {  
    public void method(int param,  
                      ServletContext cx,  
                      String s) { ... }  
}
```

- Then you will be able to access it from Javascript just as though the ServletContext parameter was not there

```
Remote.method(42,          // int param  
             "test",      // String s  
             callback);   // Callback
```

# Logging

## Logging

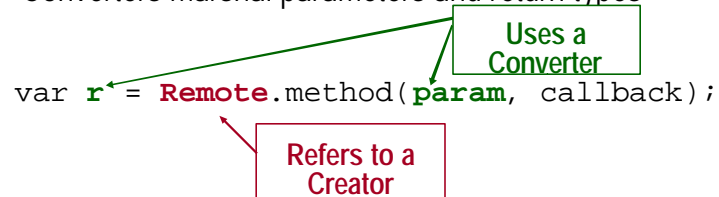
- DWR uses commons-logging if it is present
  - > java.util.logging
  - > log4j
- DWR uses `HttpServlet.log()` if commons-logging is not present

# Creators

## Creators and Converters

- Creators create objects that live on the server and have their methods remoted
- Converters marshal parameters and return types

```
var r = Remote.method(param, callback);
```



Uses a Converter

Refers to a Creator

- Created object do things while Converted objects carry data

## Advanced Creators

- Scope options
- Javascript names for session pre-population
- NullCreator for static methods
- Reloading the ScriptedCreator



# DWR Security

## Security

- DWR does not remote anything that you don't say it can via dwr.xml
- Audit
- Multiple dwr.xml Files
- Role based security
- Method level access control
- Risks

## Multiple dwr.xml Files

- For separate J2EE security domains
- Or to separate components
- Configured in web.xml:

```
<init-param>  
  <param-name>config****</param-name>  
  <param-value>WEB-INF/dwr.xml</param-value>  
</init-param>
```

## Signatures

- Sometimes introspection is not enough
- The <signature> element fixes the hole

```
<signatures>
  <![CDATA[
    import java.util.List;
    import com.example.Check;
    Check.setLotteryResults(List<Integer> nos);
  ]]>
</signatures>
```



# DWR & Web Application Frameworks

## DWR and Spring

- SpringCreator
- DwrController
- DwrSpringServlet
- SpringContainer
- beans.xml in place of dwr.xml

## DWR and Other Libraries

- StrutsCreator
- JsfCreator and FacesExtensionFilter
- PageFlowCreator
- HibernateBeanConverter
- DOM, XOM, JDOM, DOM4J
- Rife



**DWR 2.0**

## DWR 2.0

- DWR 1.1
- DWR 2.0
  - > AjaxFilters
    - > Security, Logging, Delay, Transactions
  - > Spring Integration
    - > No more dwr.xml, just use beans.xml
  - > Reverse Ajax
    - > Asynchronously push Javascript to the browser



# Introduction to DWR (Direct Web Remoting)

Sang Shin  
Java Technology Architect  
Sun Microsystems, Inc.  
[sang.shin@sun.com](mailto:sang.shin@sun.com)  
[www.javapassion.com](http://www.javapassion.com)

Welcome to the introduction to DWR (Direct Web Remoting) presentation. My name is Sang Shin from Sun Microsystems.