

**eQ!™ Scripting Engine
Developers Guide**

5/7/2003

What is the eQ!TM Scripting Engine?

The eQ!TM Scripting Engine is a component that will execute an XML based script. The XML script is 'compiled' at runtime into a set of objects that will then execute the various steps in the script. The script engine can be viewed as a short term workflow engine, however it is not meant to be a multi-input workflow engine. This means that it does not execute some operation, wait for other input and then execute some other operation. It is designed to be a series of short execution steps in support of a single user input and output. It is analogous to how a servlet engine operates. The servlet engine creates an `HttpServletRequest` and `HttpServletResponse` and gives those to a servlet that a developer writes. The servlet then uses both of these objects to inspect its parameters and attributes, execute some logic and return a response. Likewise, the script engine also has a `ScriptRequest` object that contains all of the input values for the script execution, and a `ScriptResponse` object that contains all of the output values of the script execution. The script engine is also similar in concept to how EJB Session beans are developed. Session beans are meant to be workflow logic for a portion of an application and in doing so, can invoke other session beans, entity beans, or other objects. The primary difference is that the script engine does not require java coding of the execution steps, like a servlet/JSP or EJB does. The execution steps are defined in an XML file.

Capabilities

The eQ! scripting engine has the basic requirements to fit most scripting needs. These include object creation, invoking methods on objects and doing conditional logic. However, the scripting engine is not limited by its built in abilities. It can be extended to provide custom functionality as needed. The extension capability is provided by implementing an interface and then by using a special tag in the script, the engine can then invoke the command.

Developing Scripts

The script is an XML document that gets read into the scripting engine at startup. There are only a few tags that make up the script, but there is a lot of functionality in those few tags.

Script Format

There two sections to the script file. The first is a script-config section which defines the names and types of scripts. Each script must have a unique name. A sample of this section is:

```
<configuration>
  <script-config name="add_customer" />
  <script-config name="edit_customer"
class="com.browsersoft.bc.scripting.default.DefaultScript" />
</configuration>
```

Notice that there are two variations on the `<script-config>` tag. The first just defines a script name; the second defines an implementation class. The implementation class must implement the `Script` interface. The example shown is the default implementation provided, and as such does not need to be declared.

The second section contains the actual scripts to be executed. Each script must be named and must have a corresponding name defined in the configuration section. All of the tags inside the script tag are execution steps. For example:

```
<script name="add_customer">
  <method name="preserve" stopOnError="false" target="$customer" />
  <attribute name="forward" value="success" />
</script>
```

In this example, we have the script named 'add_customer', as defined in the configuration section, and this script has two execution steps. The first step invokes a method on the object labeled 'customer' and the second step sets a response attribute named 'forward' to the value 'success'.

Values and Targets

Targets and values of the various tags follow a couple of rules. The first is that a runtime variable should be denoted by a '\$'. This tells the script engine that this is a variable name of an object that is supplied to the script. Using the '\$' notation, the dot notation, '.', may also be used to specify methods and properties on the object. The rules for this are that first the engine will check to see if the target is an eQ! component and if so, then it will try to find the specified property name using eQ!'s symbolic interface. This interface allows n-level deep property names to be used using the dot notation. If the target is not an eQ! component or if the property name is not present, then the engine will use reflection to find the method name and invoke it. If the '\$' is not present, the engine assumes that the value is a constant.

Values and Targets can be located in either the `ScriptRequest` or the `ScriptResponse`. If the notation `request.varname` is used, then the `ScriptRequest` will be searched for the `varname` variable. If the notation `response.varname` is used, then the `ScriptResponse` will be searched for the `varname` variable.

If the '\$' is not followed by 'request' or 'response', then the search proceeds in this order:

1. `ScriptResponse`
2. `ScriptRequest`

In this example, two attributes will be set in the `ScriptResponse`. The first will use the eQ! features to get the balance property of an `Account` object which is nested inside of a `Customer` object. The second method will invoke a method on the `Customer` object directly.

```
<attribute name="balance" value="$customer.checkingAccount.balance" />
<attribute name="complete_name" value="$customer.getCompleteName" />
```

Method Invocation

The `<method>` tag allows the invocation of a method on a target object. There are two required attributes for the method tag, the name of the method and the target object of the method. An optional attribute, `stopOnError`, which defaults to `false`, can be specified to notify the script that if any exception occurs while invoking the method, that the script should stop at that point and not continue executing. Methods of course can be supplied with arguments, and the scripting engine provides a means of supplying these arguments by using a nested `<arg>` tag. The `<arg>` tag has two required attributes: `value`, which is the value of the argument, and `type`, which is the class type of the argument. You can specify the full class name for the argument type, or you can use one of the following abbreviations:

- `string` = `java.lang.String`
- `int` = `java.lang.Integer`
- `long` = `java.lang.Long`
- `short` = `java.lang.Short`
- `double` = `java.lang.Double`
- `float` = `java.lang.Float`
- `object` = `java.lang.Object`

The return value from the method invocation is placed into the `ScriptResponse` under the name `'target_name.method_name'`.

The value attribute may either be a constant or a variable prefixed with a '\$'. The ordering of the `arg` tags is important and should correspond to the ordering of the method arguments.

This example will invoke the `setFirstName()` method on a customer object with a string argument of 'John':

```
<method name="setFirstName" stopOnError="true" target="$customer">
  <arg value="John" type="string" />
</method>
```

This example will invoke the `getFirstName()` method on a customer object:

```
<method name="getFirstName" stopOnError="true" target="$customer" />
```

The result of this method invocation is stored in the `ScriptResponse` under `'customer.getFirstName'`. This is an example only. A much simpler way to achieve this particular result is to use the `<attribute>` tag, described below.

Object Instantiation

The script engine can instantiate objects by using the `<bean>` tag. The tag has two required attributes, the instance name of the bean, and the class of the bean. Again, arguments can be supplied to the constructor in a similar manner as with the `<method>` tag. The name attribute is used as a key value to put the object into the `ScriptResponse` so that it may be later retrieved.

This example will create a new `Customer` object, using the constructor with two string values:

```
<bean name="customer" class="com.browsersoft.bc.example.Customer">
  <arg value="John" type="string" />
  <arg value="Doe" type="string" />
</bean>
```

Something to be aware of is that the bean may already exist. The default behavior in this case is that the script engine will not create a new instance of the object if the specified bean instance name already exists in the `ScriptResponse`. This behavior can be overridden by using the optional attribute `create="yes"` in the bean tag.

Attributes

Attributes in the output `ScriptResponse` can be set using the `<attribute>` tag. There are two required attributes on this tag, the name of the attribute and the value of the attribute.

This example will set an attribute named `'new_customer'` to a value of the object `Customer`.

```
<attribute name="new_customer" value="$customer" />
```

Conditionals

To do conditional 'if' statements, the `<if>` tag is used. There are three required attributes of this tag. The `value1` attribute is the 'left side' value of the operator being performed and `value2` is the 'right side' value. The `operator` attribute specifies the comparison to be performed. One optional attribute, the `type` attribute, tells the conditional command that a numerical comparison should be used. By default, the values are converted to strings and then compared. This can lead to invalid comparisons when numerical types

are evaluated. To overcome this, the `type` attribute must be specified and the values are then converted to the specified type before the comparison is made.

The supported numerical types are:

- `int`
- `long`
- `double`
- `float`
- `short`

The supported comparison operators are:

- `Equals`
- `GreaterThan`
- `LessThan`
- `GreaterEqual`
- `LessEqual`

If the condition is true, then the script tags, or steps, that are nested within the `<if>` tag are executed.

In this example, a customer's age is checked to see if it equals the value of 28. If this is true, the script will then set the customer's age, using the `method` tag, to the value of 29.

```
<if name="Age equality test" value1="$customer.age" operator="Equals"
value2="28">
  <method name="setAge" target="$customer">
    <arg type="int" value="29" />
  </method>
</if>
```

In the `'if'` tag, the `name` attribute is optional. However, if `'name=<name>'` is specified, then the return value of the conditional test, being true or false, is stored in the `ScriptResponse` under `'<name>.pass'`.

At this time, there is no concept of an `'else'` or `'else if'`. This will be supported in a future version. Nesting of `'if'` statements is also not currently supported.

Executing Scripts from Scripts

A script can be invoked from within a script through the use of the `<execute-script>` tag. There is one required attribute on this tag: the name of the script to be called.

This example calls `'another_script'` from within a script:

```
<execute-script name="another_script" />
```

The called script sees the same input values, in the same `ScriptRequest`, as does the calling script. The called script places its `ScriptResponse` outputs into both the `ScriptRequest` and the `ScriptResponse` used by the calling script.

Detailed Flow of Events

Putting the script engine into action takes very little Java coding, and in some cases none at all. eQ!TM provides adapters for several technologies based on J2EE, such as Struts from the Apache group, that contain all of the initialization and executing code necessary for the script engine. The following sections can be skipped by non-Java developers, or by developers who are using the eQ!TM-supplied adapters, which are described in the Appendices.

Initialization of Script Engine

The script engine only needs to be initialized once. During this initialization, the engine will read the script file and build the required scripts. The default implementation of the script engine is based on the Apache Avalon framework. The code to initialize the engine is as follows:

```
ScriptEngine engine = new DefaultScriptEngine();
DefaultScriptManager manager = new DefaultScriptManager();
DefaultConfigurationBuilder df = new DefaultConfigurationBuilder();
Configuration config = df.buildFromFile(file_location);
manager.configure(config);
((DefaultScriptManager)engine).compose(manager);
```

The `DefaultConfigurationBuilder` and `Configuration` objects are from the Avalon framework. This builder object builds a configuration based on the script XML file and supplies that to the `ScriptManager`.

The `ScriptManager` should never be directly used. It is a component used by the engine to get scripts to execute.

Executing a Script

When a developer wants to execute a script, the process is very simple and straight forward. First, a `ScriptRequest` object must be created. The name of the script is set on the request object and then any attributes that the script will be looking for are set using the `setAttribute` method. Once the request is populated with any required data, the `execute` method on the script engine is called and supplied with the request object. The `execute` method will return a `ScriptResponse` object, with which the developer can then retrieve any data that was generated by the script. The `ScriptRequest` and `ScriptResponse` objects are automatically created and used by the eQ!TM adapters for Struts and JSPs, see Appendixes A and B.

Example:

```
// Script Engine initialization and startup omitted.
ScriptRequest request = new ScriptRequest();
request.setScriptName("add_customer");
```

```

// Create some data for the script
Customer customer = new Customer();
customer.setFirstName("John");
customer.setLastName("Doe");

// Set the data into the request
request.setAttribute("customer", customer);

// Execute the engine and get our response
ScriptResponse response = engine.execute(request);

// Get data out of response as needed

```

From the Java developer's perspective, this is all that is required.

If the script generates any errors, they will be contained in the `ScriptResponse`. They are retrieved by calling the `getErrors` method which returns a collection of `ScriptExceptions`. This allows the developer to determine if any exceptions were thrown and what to do about them. This also allows exception handling in a controlled, predictable manner. The only time a `ScriptException` should be thrown from the `ScriptEngine` is if an unexpected error happened during execution. Examples are runtime exceptions, reflection exceptions, and errors. All other exceptions should be caught and captured in the response, to include user or domain specific exceptions.

Execution Steps

Every executable step, or command, in the script will generate a response that can be accessed in the response object. Depending on the step executed, the attribute key name for that response will vary. The formats are as follows:

Method Invocation

When invoking a method, the return value of that method will be under the key: `<target-name>.<method-name>`.

Example:

```
<method name="getFirstName" target="$customer" />
```

can be retrieved by:

```
response.getAttribute("customer.getFirstName");
```

Conditional

When an 'if' statement is executed, the value of the test, true or false, will be under the key: `<name>.pass`. The name is the value supplied in the name attribute of the `<if>` tag. The value will either be 'true' or 'false'.

Example:

```
<if name="Name equality 1" value1="$customer.firstName"
operator="Equals" value2="Robert" />
```

can be retrieved by:

```
response.getAttribute("Name equality 1.pass");
```

Which will return a `Boolean` object with the value of `true` or `false`;

Object Instantiation

When using the `<bean>` tag, the object will be set into the response using the name that was supplied in the `name` attribute of the tag.

Example:

```
<bean name="new_customer" class="com.browsersoft.example.Customer" />
```

can be retrieved by:

```
response.getAttribute("new_customer");
```

Extending the Script Engine

The script engine is not limited by the built in commands. Custom commands can be written to extend the script engine as needed. These custom commands are then applied to the XML script file so that the engine knows how to handle them.

Creating the Custom Command

To create the custom command, the `Command` interface must be implemented. If the default script engine is used, then the abstract class `AbstractCommand` should be extended. This abstract command provides two methods, `getCommandName` and `stopOnError`, for you. It also implements the `configure` method from which it gets configuration for the command name, target, and if it should stop on any errors, from the XML file. Sub classes should override this class as needed so that they have access to any information in their custom tag, however, be sure to call `super.configure`. Any information that the tag needs, can be specified by using attributes, or sub tags in the script file. The information from these attributes and sub tags is retrieved from the `Configuration` object which is supplied in the `configure` method when the script engine initializes. The `configure` method is only called once during the life cycle of the script.

Setting up the Script File

To register the custom command in the script file, a `<command-config>` tag must be entered in the `<configuration>` section of the script file. There are two attributes that must be set in the `<command-config>` tag. The first is the name of the tag, using the `tagName` attribute, and the second is the fully qualified class name of the command by using the `class` attribute.

For example:

```
<command-config tagName="custom" class="com.company.CustomCommand" />
```

In the `<scripts>` section of the script, the custom tag is used with its specified tag name, such as:

```
<custom name="some name" target="" stopOnError="true" />
```

Appendix A: Using the Script Engine with Struts

With Struts becoming an increasingly popular web development framework, eQ!TM provides an adapter to work with Struts and the eQ!TM Scripting Engine. This adapter is implemented as a Struts Action object, and performs all of the necessary conversions between Struts and eQ!TM. In addition to the script adapter, eQ!TM also provides an adapter for working with forms. This adapter is implemented as a FormBean object and is used to make your eQ!TM components work in a Struts form.

Invoking The Script Engine

The script engine adapter, being an Action object, is setup in the struts-config.xml file just like any other action object. In the <action> tags of the struts-config.xml file, set the type attribute to: `com.browsersoft.bc.struts.BCScriptAction`. Since the same adapter is used for all scripts, the name of the script must be supplied to the script engine. This can be done in one of two ways. The first is to set the name of the script in the bc-config.xml file as a subelement of the <form> tag. Example:

```
<form name="addCustomerForm">
  <bc name="customer" instanceName="customer" scope="request" />
  <script>add_customer</script>
</form>
```

This will cause the `add_customer` script to be executed when the form submits.

Not all Struts actions are invoked as a process of a form submission. A query parameter can be added to an Action-invoking URL to invoke the specified script to handle the Struts Action, assuming that the action has been bound to `com.browsersoft.bc.struts.BCScriptAction`. The name of the parameter is `script` and the value is the name of the script to run. Here is an example:

```
app/view.do?script=view_customers
```

This will cause the `view_customers` script to get executed, even though there is no form in use.

Information Supplied

The Struts adapter provides more than just a means of invoking the script engine. It also does all translations between the Struts/HTTP environment and the script engine environment. It will supply all application, session and request parameters, in that order, to the `ScriptRequest` object, and it will also take any business objects out of the eQ!TM FormBean adapter and supply the objects to the `ScriptRequest`. Finally, the entire HTTP Parameters hashtable will also be supplied under the variable name `'request.http-request'`. This is useful if you need to pass these parameters as a group to a method call, as is done when invoking the eQ! persistence layer.

Once the script has been executed, all objects and method returns from the `ScriptResponse` will be copied into the HTTP response, which is forwarded to the JSP that is selected for output. If you wish to put any item into the session or application scope, this can be done with the `<attribute>` tag. The name attribute of the attribute tag must be prefixed with the scope you want, such as:

```
<attribute name="session.customer" value="$customer" />
```

This will place the customer object, created elsewhere, into the session scope.

Page Forwarding

In the Struts environment, when an action is completed, the user is forwarded to some page. The script engine can supply this determination using an attribute tag with a name of 'forward'. The following example will cause Struts to forward to the success page.

```
<attribute name="forward" value="success" />
```

The 'success' value must correspond to a 'forward' entry in the `struts-config.xml` file in order to work properly. This allows screen flow logic to be scripted easily without requiring coding Struts Action java classes.

Appendix B: Invoking an eQ!™ Script from a JSP

An eQ!™ script can be invoked directly from a JSP. This can be useful in a non-Struts environment, and also from an initial page that is not the result of some previous action.

Linkage from the JSP to the eQ!™ script engine is provided by a custom tag library. In order to use this tag library in a JSP, place the following line at the start of the JSP:

```
<%@ taglib uri="/WEB-INF/struts-helper.tld" prefix="eq" %>
```

This will allow you to use the eQ!™ custom tags, prefixed by 'eq:'. Note that the name 'struts-helper' is misleading.

Our example below also uses the JSP Standard Tag Core Library. In order to use these tags you will need this line:

```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
```

Before you invoke the script, you may wish to create inputs for the script to use. For example, the following tags create a Hashtable with a retrieval parameter, that will be used by an <arg> tag in the script to pass a Hashtable of values to the retrieve() method of a BusinessObjectSet:

```
<!-- Create a bean to hold the retrieval parameters -->
<jsp:useBean id="params" scope="page" class="java.util.Hashtable" />

<!-- Put the retrieval parameters into the bean -->
<% params.put("value", "Speed"); %>
```

Use the <eq:script> tag to invoke the script. This tag creates a ScriptRequest for use by the script. Within this tag you can supply <eq:script-attribute> tags, an <eq:script-execute /> tag, and other normal JSP tags and HTML. Specify the </eq:script> after you have used any results from the script, as the created ScriptRequest and ScriptResponse go out-of-scope after this tag.

In the following example, we specify that we wish to invoke the 'retrieve_business_object_set' script. Before we execute the script we use a <eq:script-attribute> tag to supply a value to the ScriptRequest. We invoke the script with the <eq:script-execute /> tag, which results in a ScriptResponse that may be used by other enclosed JSP tags.

```
<!-- Setup to execute a script -->
<eq:script scriptName="retrieve_business_object_set">
  <!-- Setup the persisterId we want to use -->
  <eq:script-attribute name="persisterId"
    value="bySupplierName" />

  <!-- Setup the persister parameters -->
  <eq:script-attribute name="params" value="<%=params%>" />
```

```

<!-- Execute the script -->
<eq:script-execute />

<!-- Put the resulting set into the pageContext.
     This is currently required so that the JSTL
     tags can 'see' our BusinessObjectSet.
-->
<% pageContext.setAttribute( "set",
        scriptResponse.getAttribute( "bo_set" ) ); %>
<!-- Create an HTML table showing the results -->
<table>
  <tr>
    <td>Name</td>
    <td>Street</td>
    <td>City</td>
    <td>State</td>
    <td>Zip</td>
  </tr>

  <!-- Loop through the set, outputting
       the properties of each Supplier
  -->
  <c:forEach var="supplier" items="{set.iterator}" >
    <tr>
      <td><c:out value="{supplier.name}" /></td>
      <td><c:out value="{supplier.street}" /></td>
      <td><c:out value="{supplier.city}" /></td>
      <td><c:out value="{supplier.state}" /></td>
      <td><c:out value="{supplier.zip}" /></td>
    </tr>
  </c:forEach>
</table>
</eq:script>

```