An Introduction to the Bean Scripting Framework

Victor J. Orlikowski

orlikowski@apache.org

ApacheCon US - Las Vegas
November 20, 2002

What is BSF?

The Bean Scripting Framework (or, as it will be referred to for the remainder of this paper, BSF) is a set of Java classes that enables the use of scripting languages, such as Javascript and Tcl, within Java applications and that permits the use of Java objects and functions within supported scripting languages. The function provided by BSF makes it possible to write JSPs using scripting languages rather than Java (while losing none of the extensive capabilities afforded by the Java class library), as well permitting any Java application to be implemented in part (or dynamically extended) by a language that is embedded within it. This is achieved through providing an API that permits calling scripting language engines from within Java, as well as an object registry that exposes Java objects to these scripting language engines.

History

BSF began life in 1999 as a research project of Sanjiva Weerawarana at IBM's T.J. Watson research labs. The initial intent had been to provide access to Java Beans from scripting language environments (though there is nothing limiting access only to Java Beans). It was soon moved to IBM's AlphaWorks developer site, where significant interest (both internal and external to IBM) led to its being moved to IBM's developerWorks site, where BSF could operate as an Open Source project. During this time, significant development was done by Matt Duftler and Sam Ruby, and BSF was incorporated into both IBM products (Websphere) and Apache projects (Xalan). It was this interest on the part of the Apache Software Foundation that ultimately led to BSF's acceptance as a subproject of Jakarta in 2002.

During the process of moving BSF to Jakarta, development continued within IBM, with further improvements to BSF's integration with Jasper being made by

John Shin and the addition of debugging support for the Javascript language (a team effort, resulting from the work of IBM researchers Olivier Gruber, Jason Crawford, and John Ponzo and IBM software developers Chuck Murcko and Victor Orlikowski).

It is the current version, 2.3, that has been released to Apache from IBM.

Supported Languages

BSF supports several scripting languages within the standard distribution. These are:

- Javascript (provided via the Rhino engine from the Mozilla project),

- Python (provided via either the current Jython or previous JPython engines),

- Tcl (provided via the Jacl engine),

- NetRexx (a Java-based extension of the IBM REXX scripting language), and

- XSLT Stylesheets (via Jakarta's own Xalan and Xerces)

Languages that have been targeted to be added soon include:

- Ruby (provided via the JRuby engine),

- BeanShell

- JudoScript

- Perl (is often requested, but would be somewhat difficult to integrate natively - though it has been done by a project called the Perl-Java Connector)

Languages that are no longer actively maintained in this release are those using the ActiveScript engine. These include VBScript, LotusScript, and PerlScript.

Architecture

The two primary features of BSF's architecture are the BSFManager and the BSFEngine.

The BSFManager provides the overarching management of all scripting execution engines running beneath it, and maintains the object registry that permits scripts access to Java objects. Only by creating an instance of the BSFManager class does a Java application gain access to scripting services.

The BSFEngine provides an interface that must be implemented for a language to be used by BSF. This interface provides an abstraction of the scripting

language's capabilities that permits generic handling of script execution and object registration within the execution context of the scripting language engine.

In this model, an application could instantiate a single BSFManager, and access several different scripting languages in an identical manner via the BSFEngine interface. Furthermore, all of the scripting languages managed by the BSFManager are aware of the objects registered with that BSFManager, and the execution state of those scripting languages is maintained for the lifetime of the BSFManager.

Installation

BSF can be used either as a standalone system, as a class library, or as part of an application server. In order to be used as a class library or as a standalone system, one must simply download the bsf.jar file from the BSF website (http://jakarta.apache.org/bsf/index.html) and include it in their classpath, along with any required classes or jar files implementing the desired languages.

In order to use BSF as part of the Tomcat servlet engine, one must currently download patches from the BSF website that permit Jasper to call BSF. Instructions for this are posted on the website, and may soon be accompanied by pre-built binaries. It is intended that these changes will be merged into Tomcat in the near future.

Version 5 of IBM's Websphere Application Server already contains version 2.3 of BSF, while versions 3 and 4 of Websphere contain earlier versions.

Usage

JSPs and BSF

Upon having set up an application server that is BSF enabled, one can write JSPs using any of the supported scripting languages. JSPs that use scripting languages differ only slightly from those using Java.

To begin with, one must set the language attribute of the page directive in the JSP to the desired language. For example,

```
<%@ page language=''javascript'' %>
```

sets the language used for the JSP to Javascript; any scriptlets or expressions within the JSP will be handed off to BSF, which will in turn hand the code over to Rhino for execution.

Within these JSPs, the standard set of JSP implicit objects is available. These implicit objects must be used for input and output with respect to the generated page, since the scripting languages do not have any awareness of having been called within a JSP. For example, in order to print a line of text into the page

generated by the JSP, one must use the `println()` method of the `out` implicit object.

Multiple languages can be supported within a given JSP; this is accomplished by using the BSF taglibs, which are available from the Jakarta Taglibs project at http://jakarta.apache.org/taglibs/index.html. Two tags are provided: `scriptlet` and `expression`. Both of these have a required language attribute, which is used to specify the language used on a per scriptlet or expression basis.

Servlets and Applications

Using BSF in servlets or applications is also remarkably simple. In order to provide an application with scripting support, one simply needs to import the BSF class hierarchy and instantiate a BSFManager object. After having instantiated the BSFManager, one registers or declares any Java objects that one would like to make available within the scripting engine. One then calls either one of the `eval()` or `exec()` BSFManager methods (depending on whether one would like to evaluate a script and have the value of the evaluation returned, or simply execute a script). Alternatively, one can call the `loadScriptingEngine()` method in order to get an object implementing the BSFEngine interface for the desired scripting language. One can then call the `exec()` or `eval()` methods of the BSFEngine to run the script.

Furthermore, within any scripting engine's execution context, BSF declares an object named `bsf`, which represents the BSFManager that is associated with the scripting engine. This object provides all of the methods and properties associated with the BSF manager to the script. However, the most used method within scripts is usually `lookupBean()`, which is used to access objects that have been placed in BSF's object registry.

Hence, the most important methods within the BSFManager are:

- `BSFManager()` - the BSFManager constructor

- `eval()` - used to evaluate a script and return its value

- `exec()` - used to simply execute a script

- `loadScriptingEngine()` - used to return a BSFEngine for the desired scripting language

- `registerBean()` - adds an object to BSF's object registry

- `lookupBean()` - retrieves an object from BSF's object registry

- `declareBean()` - creates an implicit object in the context of any loaded scripting language, which does not have to be accessed via `lookupBean()`

Other, less often used methods within the BSFManager are:

- `apply()` - used to call anonymous functions

4

- `compileExpr()` - used to compile an expression into a CodeBuffer object

- `compileScript()` - similar to compile expression, used to compile scripts into CodeBuffer objects

- `compileApply()` - similar to both of the above' used to compile anonymous functions into CodeBuffer objects

For those that are curious, the CodeBuffer is a class provided by BSF for the storage of generated Java code.

The BSFManager `exec()`, `eval()` and `apply()` methods (as well as their compile counterparts) are simply wrappers over the equivalent methods presented by the BSFEngine interface. If one follows the path of explicitly loading a particular scripting engine via `loadScriptingEngine()`, one simply uses the `exec()` or `eval()` methods of the resulting BSFEngine as appropriate.

Incorporating Your Own Scripting Language

In order to incorporate your own scripting language into BSF, you must first write a class implementing the BSFEngine interface for your language; examples are available in the BSF source distribution.

Usually, a scripting language author simply extends the BSFEngineImpl class, which implements BSFEngine, and only requires the scripting language author to implement the `eval()` method; however, the following methods specified by the BSFEngine interface are the most commonly implemented:

- `initialize()` - used to set up the underlying scripting language engine

- `call()` - used to call functions or methods within the scripting engine

- `eval()` - used to evaluate a script

- `exec()` - used to execute a script

- `declareBean()` - used to create an implicit object within the scripting language

- `undeclareBean()` - used to remove an implicit object from the scripting language

Once you have implemented the wrapper for your language engine, you simply instantiate a BSFManager in your application, and register your engine with it via the `registerScriptingEngine()` method. Afterward, you may use your language within the application through the usual BSF semantics.

Standalone Scripts

BSF provides a facility for running scripting languages itself. Simply running

```
java com.ibm.bsf.Main
```

will produce a help message, with instructions on how to run these scripts.

Debugging

Debugging support has been added to BSF over the last year. In its current form, only debugging of Javascript in JSPs is supported. The focus has been to design an API that would permit a generic debugging framework for multiple scripting engines; however, this has remained an elusive goal. Included in the debugging support for BSF 2.3 is a rudimentary command-line debugger named jsdb, which acts as a client to a debugging server that is managed by the BSFManager. A more friendly debugger for Javascript in BSF is available as a plugin to Eclipse in IBM's Websphere Applications Developer product.