Bean Scripting Framework
(Version 2.1)

User's Guide

Sanjiva Weerawarana
IBM TJ Watson Research Center
Hawthorne, NY 10532.

December 09, 1999

http://www.alphaWorks.ibm.com/tech/bsf
mailto:sanjiva@watson.ibm.com

ABSTRACT

The Bean Scripting Framework (BSF) is an architecture for incorporating scripting into, and enabling scripting against, Java applications and applets. Using BSF, an application can use scripting, and become scriptable, against *any* BSF-supported language. When BSF supports additional languages, the application will automatically support the additional languages. Scripts in any BSF-supported language can be run directly on the command line as well.

The list of scripting languages supported by BSF is ever-growing. Both Java-implemented languages (such as Netscape Rhino and Jacl) as well as non-Java ones such as Tcl and Perl will be supported. On Win32 platforms, active scripting languages including VBScript and JScript are supported.

This version of BSF adds ActiveScript languages to BSF and also improves the core BSF APIs for compiling scripts to Java.

This document describes how to use BSF to make your application scriptable and use scripting.

TABLE OF CONTENTS

1. INTRODUCTION

The Bean Scripting Framework (BSF) is an architecture for incorporating scripting into Java applications and applets. Scripting languages such as Netscape Rhino (Javascript), VBScript, Perl, Tcl, Python, NetRexx and Rexx are commonly used to augment an application's function or to script together a set of application components to form an application.

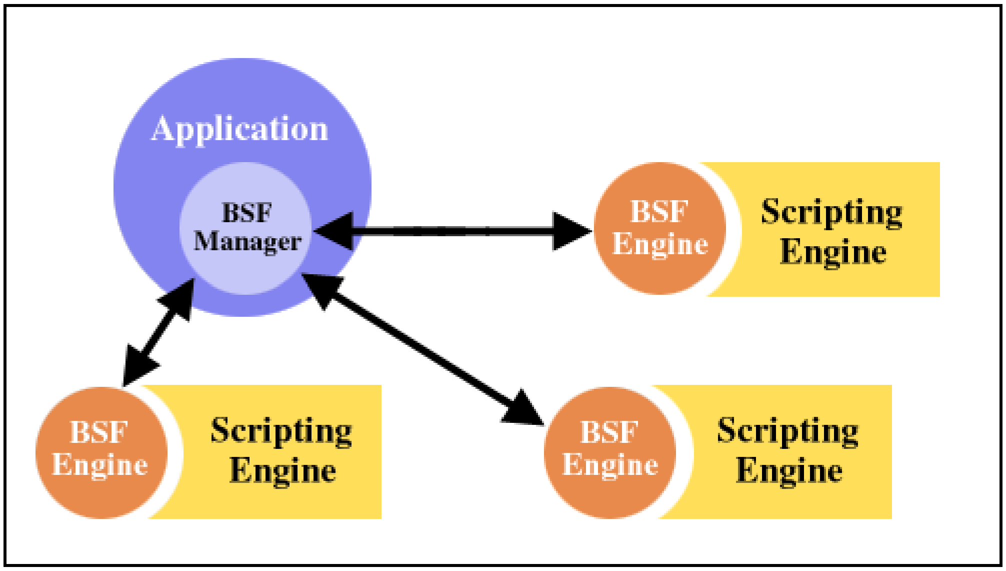
There are many scripting languages implemented in Java, including Netscape's Rhino implementation of ECMAScript, Jacl and JPython. While each of these is embeddable in Java and provides the ability to interact with Java from the language, using a specific scripting language binds an application to that single language.

The Java world currently does not have a well-defined scripting architecture that allows Java applications to incorporate scripting easily - BSF is such an architecture. The BSF architecture allows an application to be scripted from, and to script against, *any* BSF supported language, *without* any scripting language-specific modifications.

BSF supports both directions of scripting: in one case where the Java-side is in charge and runs/evaluates scripts at will, and in the other case the script runs and controls Java beans. Notice that while we use the word “bean”, it is used loosely - BSF works with any Java object and not just a true bean.

2. BSF ARCHITECTURE

The BSF architecture is composed of two basic components: a manager that provides scripting services to applications and bean services to scripting engines, and an abstraction of a scripting engine that must be supported by any BSF-enabled engine.



- BSFManager: This bean provides scripting services for the application. The manager also has a registry of known scripting engines (languages) which the application can load and interact with. A single instance of a BSFManager can manage a single instance of a language interface for any number of languages. Once a manager loads a scripting engine, it repeatedly uses that engine for all scripts evaluated by that manager in that language.
- BSFEngine: This interface must be implemented by a scripting language to become part of the bean scripting framework. The interface consists of a few methods which allow interaction with the scripting engine from the application side for running and compiling scripts. Scripts may manipulate new beans as well as any beans made available for scripting by the application.

If the scripting engines are implemented in Java, then all of the components are contained within a JVM. However, if they are in other languages, then the implementation of the BSFEngine interface would be split between Java and non-Java with a JNI link between them and the scripting engine itself would be outside of the JVM.

2.1 BSF FUNCTIONALITY

What does it mean to be able to call into a scripting language from Java? You can:

- call a method (procedure) and get the return value, and
- evaluate some piece of script (say an expression) and get its return value
- execute some piece of script for the purposes of its side-effects
- compile a piece of script or an expression into Java code

To be interesting, the script must be able to access and modify some beans in your Java application. BSF supports this by allowing you to register beans into the BSFManager's bean registry. Then, scripts can (in a scripting language dependent way), get at pre-registered beans and manipulate them by doing all the usual bean things - setting and getting properties and fields, calling methods, doing event bindings as well as creating new beans. All bean operations are done in the Java VM and the result is returned into the scripting side in a manner compatible with the scripting language.

For Java-based scripting languages like Netscape Rhino, this works very well because all Java types come down naturally into the scripting environment. For non-Java languages like Tcl and Perl, you have to use BSF provided accessor functions to manipulate Java objects from within the scripts.

What can you do from a scripting language? You can:

- look up a pre-registered bean and access a pre-declared bean
- register a newly created bean so that the Java side can get at it,
- do all bean operations (create new beans, modify them, do event bindings and call methods), and
- bind events to scripts in the scripting language.

BSFManager supports running scripts directly, i.e., a script that creates beans and performs some function can be run by itself either from the command line or within the JVM. This allows BSF scripts to become first class executables.

2.2 REGISTERING AND DECLARING BEANS

In order for scripts to manipulate Java beans that the application wishes to expose, the application must either register or declare the bean into the BSFManager. When the application registers a bean (i.e., some Java object) by giving it a name, the bean is placed in an object registry which is essentially a hashtable with the name as the key. Scripts may then look up objects by using the name of the object as the key. Thus, registering beans is a way in which the application may expose a set of beans which scripts may look up and use on an as-needed basis.

BSF supports a second way to expose beans to scripts: declaring beans. The difference between declaring and registering is that engines are expected to make declared beans “pre-available” in the scripts as far as possible. That is, if a script author needs a registered bean, (s)he needs to look it up in some way. However if (s)he needs a declared bean, the engine has the responsibility to (attempt to) make those beans available “automatically” by the same name (preferably) within the scripting language. That is, if a bean named “potato” is declared to a manager, then say a Javascript script running in that manager would have access to a variable named “potato” which is a reference to the Java object declared with the name “potato.” *Because engines are strongly urged to make declared beans available under the same name, applications are strongly urged to use a small character set when naming declared beans.* We recommend using names that correspond to this regular expression: [a-z][a-z0-9_]*.

In BSF 2.1, when a bean is declared it is automatically registered as well. Hence any declared bean can be accessed by looking it up as well. Thus, engines which choose not to implement declared beans may do so safely assuming that the declared beans can be accessed anyway. Note that BSF requires the type of declared beans to be given at the time of declaration. This is needed to enable cases where the declared type is say an interface and the object’s actual type is potentially a non-public implementation class.

When a bean is declared into the BSFManager, that is noted in the manager's internals and then forwarded onto any already running engines. If any of the already running language engines throw an exception when asked to declare the bean, then the manager's declare bean method will forward on that exception. That is, any engines that come afterwards in the engine enumeration will not even be told about this new bean. Thus, in general it is best to declare beans before the manager has been asked to load any engines because then the user can be informed when an engine rejects it. Also, its much more likely that an engine can declare a bean at start time than it can at any time.

2.3 INTERPRETED AND COMPILED LANGUAGES

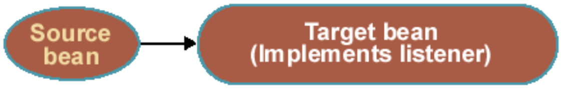
BSF supports both interpreted scripting languages (such as Netscape Rhino) as well as compiled ones (such as NetRexx). When a compiled language engine is asked to evaluate or execute a script, it compiles the script the first time its seen and executes it. Future calls to execute the same script do not need to be compiled. When asked to compile a script or expression, such a language would generate customized Java code that when compiled and run would produce the desired results. When an interpreted language (such as Jacl) engine is asked to compile a script, it generates Java code which uses BSF to evaluate the script.

2.4 BINDING EVENTS TO SCRIPTS

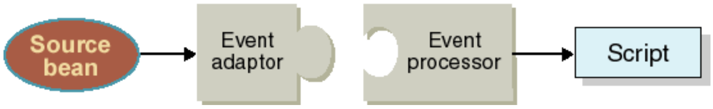
For scripting languages that need assistance in binding scripts to be executed when events occur in some bean, BSF provides an event binding service. The language-specific documentation indicate whether event bindings to scripts are done this mechanism or a language provided mechanism.

The BSF mechanism is supported by event adapters, event processors and an event adapter registry. Basically, the model consists of an event-type specific adapter that receives the event from the source, forwards it to a generic event processor which then executes the script. Event-type specific event adapters are located from the event adapter registry. The figure below illustrates the standard and extended binding architecture.

Conventional Event Binding



Non-conventional Event Binding



Event adapters must implement the `com.ibm.cs.event.EventAdapter` interface to be part of the BSF event architecture. This interface is defined as follows:

```
public interface EventAdapter {
  public void setEventProcessor (EventProcessor eventProcessor);
}
```

A base implementation of this interface is available in `com.ibm.cs.event.EventAdapterImpl` that event adapters may choose to extend. The code for EventAdapterImpl is as follows:

```
public class EventAdapterImpl implements EventAdapter {
      protected EventProcessor eventProcessor;

      public void setEventProcessor (EventProcessor eventProcessor) {
        this.eventProcessor = eventProcessor;
      }
    }
```

For each type of event (actually, for each listener type), an event adapter must be implemented and available from the event adapter registry. For example, the following event adapter is included in the BSF jar file for adapting java.awt.event.KeyEvent events to the BSF event architecture:

```
    public class java_awt_event_KeyAdapter extends EventAdapterImpl
                                          implements KeyListener {
      public void keyTyped (KeyEvent e) {
        eventProcessor.processEvent ("keyTyped", new Object[]{e});
      }
      public void keyPressed (KeyEvent e) {
        eventProcessor.processEvent ("keyPressed", new Object[]{e});
      }
      public void keyReleased (KeyEvent e) {
        eventProcessor.processEvent ("keyReleased", new Object[]{e});
      }
    }
```

When the BSF runtime creates event adapters and adds them as listeners to the event sources, it tells the adapter what event processor to forward (delegate) events to. Event processors are the entry point to the BSF runtime and are responsible for delivering the event to the intended recepient script. The `com.ibm.cs.event.EventProcessor` interface is defined as follows:

```
    public interface EventProcessor {
      public void processEvent (String filter, Object[] eventInfo);
      public void processExceptionableEvent (String filter, Object[] eventInfo)
                                            throws Exception;
    }
```

When an event adapter receives an event from an event source, it delegates the event to its event processor using one of the above two methods. If the event method may throw an exception, then it should delegate it via the `processExceptionableEvent` method, otherwise via the `processEvent` method. The filter argument is in general the name of the method via which the event was received. The exception to this is for `java.beans.PropertyChangeListener` and `java.beans.VetoableChangeListener` event listener types, where the filter is the name of the property. The BSF event support class uses a single event processor that actually delivers the event to a script and runs the script.

The *event adapter registry* (`com.ibm.cs.event.EventAdapterRegistry`) provides a static registration and look up service for event adapters. The EventAdapterRegistry class has the following two methods:

```
    public static void setClassLoader (ClassLoader cloader)
    public static void register (Class listenerType, Class eventAdapterClass)
    public static Class lookup (Class listenerType)
```

This implementation uses a simple hashtable as its registry. The BSF jar file contains adapters that the above registry is aware of for the following listener types:

- all `java.awt.event.*` event listener types
- `java.beans.PropertyChangeListener` and `java.beans.VetoableChangeListener` event listener types

NOTE: In BSF v2.1, the arguments of the event listener method are not available to the scripts. This will be fixed in a later release.

If an application needs to enable scripts to bind to other events, then the appropriate adapter must be written and registered with the event adapter registry.

3. INCORPORATING BSF IN AN APPLICATION

Incorporating BSF in an application is easy. The basic process is to create an instance of a BSFManager and then load the appropriate scripting languages into it. The application can register any beans it wants to make available for scripting into the bean registry of the manager. When engines run scripts, they can access these beans and interact with them and do whatever the script needs to do.

You can have multiple BSFManager beans in your application. The motivation for that would be if you wanted to have multiple instances of scripting languages running at the same time within the same JVM. In most cases that would not be the case and you would have one manager per application.

Here are the steps to successful BSF integration:

1. Declare and create a BSF manager:

```
import com.ibm.bsf.*;

class YourClass {
  BSFManager mgr = new BSFManager ();

  ...
}
```

2. Register any new scripting languages:

```
String[] extensions = {”pos”};
mgr.registerScriptingEngine (“potatoscript”,
                             edu.purdue.cs.bsf.engines.potatoscript,
                             extensions);
```

3. Register or declare any beans you want to make available to scripting engines with appropriate names:

```
mgr.registerBean (“myFrame”, myFrame);
mgr.registerBean (“myOtherBean”, myOtherBean);
mgr.declareBean (“respose”, response, Response.class);
```

   Note that *any* Java object (i.e., any bean) can be made available for scripting. There is no need for the object to implement any specific interface etc..

4. (Optionally) Load any scripting engines that you want:

```
BSFEngine rhinoEngine = mgr.loadScriptingEngine (“javascript”);
```

   Now you’re ready to call functions in the script or to evaluate/execute arbitrary scripts:

```
Object result = rhinoEngine.eval (“testString”, 0, 0, “2+32”);
result = rhinoEngine.call (null, “hello”, null);
rhinoEngine.exec (“testString2”, 0, 0, “hello ()”);
```

Alternatively, you can have the manager run any scripts directly:

```
Object result = mgr.eval (“javascript”, “testString”, 0, 0, “2+32”);
mgr.exec (“javascript”, “testString”, 0, 0, “hello ()”);
```

Or you can compile scripts or expressions into Java code:

```
import com.ibm.cs.util.CodeBuffer;

CodeBuffer cb = new CodeBuffer ();
rhinoEngine.compileScript (“testString”, 0, 0, “hello ()”, cb);
String res1 = cb.toString ();
cb = new CodeBuffer ();
mgr.compileScript (“javascript”, “testString”, 0, 0, “hello ()”, cb);
String res2 = cb.toString ();
```

In either case, res1 and res2 are:

```
public class Test
{
  com.ibm.bsf.BSFManager bsf = new com.ibm.bsf.BSFManager();

  public java.lang.Object exec() throws com.ibm.bsf.BSFException
  {
    return bsf.eval("javascript", "testString", 0, 0, "hello ()");
  }
}
```

(See the API docs for BSFManager and BSFEngine for details.)

The scripts you are evaluating can access the beans registered in the BSF registry using accessor functions in a language dependent manner. This mechanism is described in the next section.

4. RUNNING BSF SCRIPTS DIRECTLY

BSF scripts can be run directly on the command line using a utility class provided in BSF:

```
% java com.ibm.bsf.Main
Usage:

  java com.ibm.bsf.BSFManager [args]

    args:

      [-in            fileName]   default: <STDIN>
      [-language  languageName]   default: <If -in is specified and -language
                                            is not, attempt to determine
                                            language from file extension;
                                            otherwise, -language is required.>
      [-compile       (on|off)]   default: off

    Additional args used only if -compile flag is set to "on":

      [-out          className]   default: Test
```

5. INTERACTING WITH JAVA FROM A SCRIPTING LANGUAGE

The BSF architecture allows a script to access Java and manipulate Java beans. The methodology for doing this is scripting language-specific and is documented below for each currently supported language.

6. LANGUAGE-SPECIFIC INFORMATION

This section provides language-specific information that application authors and script authors need to be aware of. Version 2.1 of BSF supports the following languages (with more under active development):

- Mozilla Rhino v1.4 Release 3
- NetRexx v1.148 upwards
- BML v2.4
- JPython v1.1-beta3
- Jacl v1.1.1
- LotusXSL v0.18.*
- Microsoft ActiveScript languages VBScript, JScript and ActiveState's PerlScript.

6.1 MOZILLA RHINO (JAVASCRIPT) V1.4 RELEASE 3

Category	Description
Language identifier	javascript
File extension(s)	.js
Looking up a bean	x = bsf.lookupBean (“name-of-bean”)
Create a new bean	uses Rhino’s mechanisms
Registering a bean	bsf.registerBean (“name-of-bean”, bean)
Unregistering a bean	bsf.unregisterBean (“name-of-bean”)
Binding a script to be executed upon event firing	bsf.addEventListener (bean, "event-set-name”, “filter”, "script-to-exec”) where event-set-name is the name of the event, filter is the specific method of the event’s listener interface via which event must be received to be delivered to the script (may be null), and script is the script to be exec’ed when the event occurs.
Doing all bean operations	Uses Netscape’s LiveConnect technology; see `http://www.mozilla.org/js/liveconnect`
Language home	`http://www.mozilla.org/rhino/`

6.2 NETREXX V1.148 UPWARDS

Category	Description
Language identifier	netrexx
File extension(s)	.nrx
Looking up a bean	x = bsf.lookupBean(“name-of-bean”) Note that NetRexx does not allow whitespace between the function name and the parenthesis!
Create a new bean	uses NetRexx’s mechanisms
Registering a bean	bsf.registerBean(“name-of-bean”, bean)
Unregistering a bean	bsf.unregisterBean(“name-of-bean”)
Binding a script to be executed upon event firing	Use NetRexx’s mechanisms.
Doing all bean operations	NetRexx is in many ways an alternate syntax for Java. Java operations can be done directly in NetRexx.
Language home	`http://www2.hursley.ibm.com/netrexx/`

6.3 BML v2.4

Category	Description
Language identifier	bml
File extension(s)	.bml
Looking up a bean	<call-method target=”bsf” name=”lookupBean”> <string value=”name-of-bean”/> </call-method>
Create a new bean	uses BML’s mechanisms
Registering a bean	<call-method target=”bsf” name=”registerBean”> <string value=”name-of-bean”/> bean-to-register </call-method>
Unregistering a bean	<call-method target=”bsf” name=”unregisterBean”> <string value=”name-of-bean”/> </call-method>
Binding a script to be executed upon event firing	Use BML’s mechanisms.
Doing all bean operations	Use BML’s mechanisms.
Language home	`http://www.alphaWorks.ibm.com/formula/bml`

6.4 JACL v1.1.1

Category	Description
Language identifier	jacl
File extension(s)	.jacl
Looking up a bean	bsf lookupBean “bean-name”
Create a new bean	uses Jacl’s mechanisms
Registering a bean	bsf registerBean “bean-name” $bean
Unregistering a bean	bsf unregisterBean “bean-name”
Binding a script to be executed upon event firing	bsf addEventListener $bean "event-set-name" "filter" "script" where event-set-name is the name of the event, filter is the specific method of the event’s listener interface via which event must be received to be delivered to the script (may be empty), and script is the script to be exec’ed when the event occurs.
Doing all bean operations	Use Jacl’s mechanisms.
Language home	`http://www.scriptics.com/products/java/`

6.5 JPYTHON V1.1-BETA3

Category	Description
Language identifier	jpython
File extension(s)	.py
Looking up a bean	bsf.lookupBean (“name-of-bean”)
Create a new bean	uses JPython’s mechanisms
Registering a bean	bsf.registerBean (“name-of-bean”, bean)
Unregistering a bean	bsf.unregisterBean (“name-of-bean”)
Binding a script to be executed upon event firing	Use JPython’s mechanisms.
Doing all bean operations	Use JPython’s mechanisms.
Language home	`http://www.jpython.org/`

6.6 VBSCRIPT (ON WIN32 ONLY)

Category	**Description**
Language identifier	vbscript
File extension(s)	.vbs
Looking up a bean	bsf.lookupBean ("name-of-bean")
Create a new bean	set b = CreateBean ("type-of-bean") or set b = bsf.createBean ("type-of-bean")
Registering a bean	bsf.registerBean ("name-of-bean", bean)
Unregistering a bean	bsf.unregisterBean ("name-of-bean")
Binding a script to be executed upon event firing	var.on<EventName> = script-to-exec
Doing all bean operations	method calls: *var.methodname args* (same syntax as typical VB, method names are case-insensitive as much as possible) property setting: *var.propertyname = value* property getting: *var.propertyname* field access is not supported
Language home	`http://msdn.microsoft.com/scripting`

6.7 JSCRIPT (ON WIN32 ONLY)

Category	**Description**
Language identifier	jscript
File extension(s)	.jss
Looking up a bean	bsf.lookupBean ("name-of-bean")
Create a new bean	var b = CreateBean ("type-of-bean") or var b = bsf.createBean ("type-of-bean")
Registering a bean	bsf.registerBean ("name-of-bean", bean)
Unregistering a bean	bsf.unregisterBean ("name-of-bean")
Binding a script to be executed upon event firing	var.on<EventName> = script-to-exec
Doing all bean operations	method calls: *var.methodname (args)* (same syntax as typical JScript) property setting: *var.propertyname = value* property getting: *var.propertyname* field access is not supported
Language home	`http://msdn.microsoft.com/scripting`

6.8 PERLSCRIPT (ON WIN32 ONLY)

Category	**Description**
Language identifier	perlscript
File extension(s)	.pls
Looking up a bean	$bsf->lookupBean ("name-of-bean")
Create a new bean	$b = CreateBean ("type-of-bean") or $ = $bsf->createBean ("type-of-bean")
Registering a bean	$bsf->registerBean ("name-of-bean", bean)
Unregistering a bean	$bsf->unregisterBean ("name-of-bean")
Binding a script to be executed upon event firing	$var->on<EventName> = script-to-exec

Doing all bean operations	method calls: *$var->methodname (args)* (same syntax as typical Perl) property setting: *$var->propertyname (value)* property getting: *$var->propertyname* field access is not supported
Language home	`http://www.activestate.com`

7. EXAMPLES

The demos directory in the distribution has some examples of using BSF.

The above examples are small examples that illustrate the basics of BSF usage. The following systems use BSF today to satisfy their scripting needs:

- BML (`http://www.alphaWorks.ibm.com/formula/bml`) for implementing <script> elements in any scripting language.
- LotusXSL (`http://www.alphaWorks.ibm.com/tech/LotusXSL`) for implementing the extension framework.
- IBM WebSphere v3.0 upwards (`http://www.software.ibm.com/websphere`) uses BSF to get multiple language JSPs.