Adaptor Replacement for Build Forge 2011

November 11, 2010 - for BF2011 M6 – initial release
January 21, 2011 – for BF2011 M7 – added BOM functions

Important Disclaimer*: this document is intended to give customers preliminary information on the current state of development of the Build Forge 2011 release. Nothing in this document is warranted. In addition, no commitment is made, express or implied, that any of the functionality described will be in the final release or will work as described.*

As of the 2011 Build Forge product, the old XML-based Adaptor functionality will be completely removed and replaced.

Issues with XML Adaptors

The old Adaptors had a long list of isses:

- slow
- confusing and limited XML syntax
- no validation for the XML - adaptor XML used a recursive structures that the XML specifications do not allow
- difficult to debug and support

Adaptors were designed for a special purpose: integration with external applications, like source control systems. However, they provided little functionality that a normal step couldn't accurately reproduce. There were only a few features that could take place only within an Adaptor.

Replacement for XML Adaptors: JavaScript Steps

The Adaptor XML functionality is being replaced by the following functionality:

- JavaScript steps - steps that run JavaScript code
- JavaScript functions - functions for JavaScript that replicate certain adaptor functionality
- Changed UI - support for XML-based adaptors will be removed. Some additions will be made within the Steps panels.
- Migration scripts - scripts that convert old adaptors and adaptor templates into JavaScript steps.

Adaptor functionality is easily re-engineered within the JavaScript step system.

Customer Use Cases

Build Forge has shipped Adaptor Template sample files with previous versions. Customers have built adaptors and their own adaptor templates from them.

- New customers will use the provided JavaScript implementations. We will not ship the xml files.
- Existing customers will need to migrate their BF-provided samples and their own adaptor templates and adaptors to JavaScript steps.

Functionality List as of 1/2011

- JavaScript steps – ready
- JavaScript functions - ready. Added BOM functions to the set released with M6
- Changes in the UI - not ready for test
- Migration Scripts - not ready for test

Each item in the list is described in more detail in sections below.

A full example is shown in its own section at the end.

JavaScript Steps

JavaScript steps have been implemented. They are represented as a step type in the UI.

JavaScript steps in general give a vast amount of added functionality to the customer. A single JavaScript step could be a kind of super-JPO. JavaScript allows for more complex flow control than JPO, including unlimited amounts of loops within loops within loops, conditional execution, temporary variable creation, and more. Supporting adaptor-equivalent behavior is just one use for JavaScript steps.

The new JavaScript step is created as a step. The step can be made a JavaScript step in one of two ways:

- When creating or editing the project, set the Default Step Provider to be "JavaScript Step". Any steps within the project that have their Step Provider set to "-- Default --" will be executed as JavaScript steps.
- When creating or editing a particular step, set the Step Provider to be "JavaScript Step". Only this step will be a JavaScript step.

Any code in the "Command" block is interpreted as JavaScript code to be executed.

The code must start with a step_main function. It is provided by the user.

All other step settings should work normally.

Limitations on JavaScript Support

JavaScript steps support what is known as "core JavaScript." It does not support the Document Object Model (DOM). The two are often used together to drive web pages. JavaScript within a JavaScript Step is used strictly as a programming language - it does not execute in a browser context. It executes in the context of a running Build Forge job. It is executed by MJC.

Any code that relies on an existing browser is not supported. This includes all of the document and window functions (document.write, window.location, and so forth), alert(). UI elements (browser objects) are not supported (Window, Navigator, Screen, History, Location).

We provide functions to allow the JavaScript steps to perform input and output operations. They are used instead of UI- or browser-related constructs that perform the same function. For example, the JavaScript print functions do not print to the step log. You use a provided function to write to the log.

JavaScript is supported by Mozilla. A reference of core JavaScript elements is here:
https://developer.mozilla.org/en/JavaScript/Reference

JavaScript functions

All JavaScript functions we provide are within the `BF_Obj_JavaScriptStep` object.

Quick List

This list is presented in the same order as the reference.

Groups and Users
- addUserToTempGroup - add user to a group, creating the group if necessary
- artifactRelatedToUser - true if artifact is related to the specified user
- groupContainsUser - true if the specified user is in the specified group
- relateArtifactToUser - relate a specified file to a user as an artifact

Print to step log
printToLog - print a message to the step log

Environment
setBuildEnvironment - set variable to value
getBuildEnvironment - get value of variable
setEnvironment - set value of variable within a specified group

Command Handling
executeCommand - execute a specified command
getCommandResponseLine - get a line of command response output

Set Step Status
failStep - set step status to fail (does not affect job execution)
passStep - set step status to pass (does not affect job execution)

Notification
setNotification - sets conditions, recipients, subject, and message for a notification

Sleep
sleep - stop execution for a specified number of seconds

Job Control
requeueBuild - restart the job after a specified wait time
purgeBuild - run a purge on the job (uses Classes)
destroyBuild - remove the build completely and reset the system tags
terminateStep - immediately end job processing as passed or failed

BOM Control
addBomCategory – adds a BOM category
addBomSection – adds a section to an existing category
addBomFields – adds fields to an existing section
addBOMRow – adds a row of data to the BOM (in field creation order)
addBomFieldData – adds a row of data to the BOM (in any order)

Function Reference

`BF_Obj_JavaScriptStep.addUserToTempGroup(groupname, username)`

Adds the user *username* to the temporary group *groupname*. If the *groupname* group does not exist, it is created. This is intended to create a notification group for emailling.

The *username* may be either the login name of a Build Forge user or a direct email address. Groups created by this function do not last outside the step in which they are created.

`BF_Obj_JavaScriptStep.groupContainsUser(groupname, username)`

Returns true if the user *username* belongs to the temporary group *groupname*.

`BF_Obj_JavaScriptStep.relateArtifactToUser(artifact, username)`

Relates *artifact* to *username*. An *artifact* is a file name. A *username* is a Build Forge login name or direct email address

This function is used for the Notify Changers log filter pattern action. The intended use case is for source control Adaptors - the Adaptor gets a list of files that have been changed and the users

who changed them since the last build from the source control repository, and then later in the build when a failure happens with a particular file, such as "Error on line 47 of filename.java", the build knows that filename.java was last edited by the user "bob" and sends "bob" an email.

Note: *the Notify Changers functionality is not completely implemented as of this writing - it currently logs a build note rather than sending email to the user.*

`BF_Obj_JavaScriptStep.artifactRelatedToUser(artifact, username)`

Returns true if the artifact *artifact* is related to the user *username*.

`BF_Obj_JavaScriptStep.printToLog(line)`

Prints *line* to the Build Forge step logs. Do not use JavaScript print functions.

`BF_Obj_JavaScriptStep.setBuildEnvironment(name, value, type, placement)`

Sets the value of the build environment variable *name* to *value* within the build environment for the current build.

The *type* is an optional setting that can be any of these values:

- ENVVAR_TYPE_SET

(Default) Sets the build environment variable normally

- ENVVAR_TYPE_SET_IF_NOT_SET

Sets the build environment variable only if it was not previously set

- ENVVAR_TYPE_APPEND

Appends to the build environment variable (currently uses the ':' UNIX-type PATH separator)

- ENVVAR_TYPE_PREPEND

Prepends to the build environment variable (currently uses the ':' UNIX-type PATH separator)

The *placement* is an optional setting that can be any of these values:

- ENVVAR_PLACE_APPEND

(Default) Creates the build environment variable as a new build environment variable at the end of the build environment. This placement should not change previously existing build environment variable values.

- ENVVAR_PLACE_MODIFY

Modifies all existing build environment variables with this name. If no such variables exist, this placement acts as ENVVAR_PLACE_APPEND.

- ENVVAR_PLACE_PREPEND

Creates the build environment variable as a new build environment variable at the beginning of the build environment. This placement might change later variables whose values rely on the variable.

`BF_Obj_JavaScriptStep.getBuildEnvironment(name)`

Gets the current value of the build environment variable *name*.

`BF_Obj_JavaScriptStep.setEnvironment(groupname, name, value, type, placement)`

Sets the current value of the environment variable *name* in the environment group *groupname* to *value*. Uses identical *type* and *placement* parameters as BF_Obj_JavaScriptStep.setBuildEnvironment(name, value, type, placement).

`BF_Obj_JavaScriptStep.executeCommand(command)`

Executes the command *command* on the agent that the JavaScript step is running on. Returns

true if the command succeeded (that is, had a return code of 0) and false otherwise. The command output is visible via BF_Obj_JavaScriptStep.getCommandResponseLine().

`BF_Obj_JavaScriptStep.getCommandResponseLine()`

Gets the next command response line from the most recent `BF_Obj_JavaScriptStep.executeCommand() call.` You must use this function in a loop to iterate over multiple lines returned by a command.

`BF_Obj_JavaScriptStep.failStep()`

Sets the current step result to "Failed." This does not stop step processing - later calls to BF_Obj_JavaScriptStep.passStep() may change the step result.

`BF_Obj_JavaScriptStep.passStep()`

Sets the current step result to "Passed," clearing the fail status. This does not stop step processing - later calls to BF_Obj_JavaScriptStep.failStep() may change the step result.

`BF_Obj_JavaScriptStep.setNotification(onpass, groupname, subject, body)`

Creates a notification email that is sent when the build completes. The parameters are:

- *onpass*

If true, this notification message is sent when the build completes successfully or with warnings. If false, this notification message is sent when the build fails to complete.

- *groupname*

The name of the group to which to send this email message. The group can be either a temporary group created by addUserToTempGroup() or a permanent Rational Build Forge group. The function uses the group definition in effect when the function starts. Users added to the group after the function starts do not receive the notification message. Note that if a temporary group has the same name as a permanent Build Forge group, the permanent Build Forge group will be ignored - only the temporary group will be used.

- *subject*

The subject line to use for the email message.

- *body*

The body of the email message to be sent.

`BF_Obj_JavaScriptStep.sleep(seconds)`

Causes the JavaScript step to sleep, or pause processing, for *seconds* number of seconds. Similar to the .sleep dot command.

`BF_Obj_JavaScriptStep.requeueBuild(requeueIn)`

Causes the build to destroy itself (using the BF_Obj_JavaScriptStep.destroyBuild() function), and then requeue itself to be run later in *requeueIn* number of seconds.

`BF_Obj_JavaScriptStep.purgeBuild()`

Causes the build to purge itself when it completes. This purge uses the normal class rules for the class that the build belongs to, runs purge chains, and the like.

`BF_Obj_JavaScriptStep.destroyBuild()`

Causes the build to destroy itself when it completes. This destruction ignores the normal class rules, runs no purge chains, and removes all data for the build, as well as dropping auto-increment tag variables for the build's project.

`BF_Obj_JavaScriptStep.terminateStep(message, passfail)`

Causes the JavaScript step to end processing immediately. *message* is printed to the step logs, and the step completes with a "passed" status if *passfail* is true or a "failed" status otherwise.

`BF_Obj_JavaScriptStep.addBomCategory(name)`

Adds the BOM category name to the BOM. An existing category is required for all other BOM functions.

`BF_Obj_JavaScriptStep.addBomSection(categoryName, sectionName, parentSectionName)`

Adds the top-level section *sectionName* to the category *categoryName*. To make it a subsection of an existing section, specify the existing section as *parentSectionName.*

`BF_Obj_JavaScriptStep.addBomFields(categoryName, sectionName, field1, field2 ...)`

Adds a list of fields to section *sectionName*. If the section does not exist, creates it.

Currently you can run this function to add fields to a given section only once.

`BF_Obj_JavaScriptStep.addBomRow(categoryName, sectionName, value1, value2 ...)`

Adds a row of data to the BOM in category *categoryName* and section *sectionName*. It adds the values to the section fields in the order specified when the fields were created: the first value is assigned to the first field, the second value is assigned to the second field.

`BF_Obj_JavaScriptStep.addBomFieldData(categoryName, sectionName, fieldn, valuen, fieldm, valuem ...)`

Adds a row of data to the BOM in category *categoryName* and section *sectionName*. The fields are specified by name and can be specified in any order.

Known issue in M7: writing the final field in a row writes the row, regardless of where the final appears in the function. This will be addressed.

Changes to the UI

(In development – not released)

The existing UI infrastructure around old-form Adaptors needs a number of changes in order to bring it into parity with the current implementation. The changes that need to be made are:

Name Changes

The name "Adaptor" may be changed altogether to
a) emphasize that the JavaScript step functionality is much broader than the previous Adaptor functionality
b) emphasize that the way that Adaptors function is now completely different
c) move away from people's confusion about the Adaptor / Adapter spelling. The final naming convention hasn't yet been determined - suggestions are welcome. Stored Steps is currently in play.

Rename Projects > Adaptors

This will likely be renamed to Projects > Stored Steps (or whatever we decide). This will be an area where the stored steps can be managed.

Remove Projects > Adaptor Links

Adaptor Links are no longer necessary. Since Adaptors are now implemented as a standard type of step, with all the configuration options that that implies, there's no need any longer for a special way of linking an Adaptor with a project.

Remove Projects > Templates

There was an artificial separation between adaptors provided by Build Forge, which we called "Adaptor Templates", and adaptors created by the customer, which we called "Adaptors". This was unneccessary split that complicated our code internally, as we needed to allow customers to select either type of object for their Adaptor usage, to create Adaptors based on Adaptor Templates, etc. This split will be removed, the old Adaptor Templates will become Adaptors (or Stored Steps, or whatever we call them), and customers will be able to create Adaptors based on them in the normal way, by use of the already-existing "Copy" functionality within the UI.

Remove Related System Parameters

Previously, there were a number of System parameters that related to the Adaptor functionality, including the **Link Manual Build** and **Link Debug Mode** parameters. These are unnecessary and will be removed - the **Link Debug Mode** will be replaced by the **Visible** step setting - see below - and the **Link Manual Build** setting will be replaced by the standard step selection on build start.

Add Step Property: Visible

Previously, Adaptors that were not in Debug mode created steps that were not necessarily visible to users who viewed the build afterward. If the Adaptor step passed, the step wasn't shown within the UI, and if the Adaptor step failed, the build was purged. (System parameter settings, Adaptor Link settings, and usage of the .source and similar dot commands could modify this behavior.) Instead of this, we will be providing a **Visible** property for steps, whether or not a step is a JavaScript step. If the **Visible** property is set to Yes, the step will be visible when the build is run, as per the normal operation. If it is set to No, the step will not be visible within the UI when the build is run.

Adaptor Selection

There will be a mechanism for allowing a step to select an existing Adaptor (called JavaScript Stored Step below):

- directly: the step calls the JavaScript Stored Step. It is not visible in the step. This allows multiple steps to use a particular JavaScript Stored Step. If the stored step is modified, all steps using it are affected.
- template: the JavaScript Stored Step code is copied into the step's Command section. The user can edit it as needed.

This mechanism has not entirely been determined. It may include widgets like the following:

- **JavaScript Stored Step** pulldown - provides a list to choose from
- **Use JavaScript Stored Step Directly** button - controls whether the stored step is used directly or copied into the step.

Adaptor-related Dot Commands

.source, .test, .defect, .interface
The dot commands that reference Adaptors haven't yet been implemented. Since they all have identical

functionality, they will probably be combined into a single dot command.

Adaptor Migration

(In development - not in M6)

Migration code turns existing XML Adaptors into JavaScript Steps. This code is executed during an upgrade to the new version of BuildForge. The intention is that this functionality is transparent to the customer.

They start with their old-style XML Adaptors. After upgrade their Adaptor Templates and Adaptors are shown as JavaScript Steps, with "Visible" set where appropriate.

This is probably the code most likely to fail in the field. There are always special cases for particular customer Adaptors that they've changed according to our documentation. This should require a fair amount of testing.

Example Of JavaScript Step Code

Note: this example code runs only on UNIX or Linux systems.

The following JavaScript code snippet shows how to use some of the functions we provide, as well as how to use some of the JavaScript language functionality.

This example does the following:
1. goes to a particular directory (contained in the $TESTPATH build environment variable)
2. lists the files
3. looks for filenames that end in .txt or .java and have no spaces
4. prints those filenames to the step log.

If the file necessary.java is not found:

1. sends a failure email to the Build Forge "Developer" access group
2. queues the build to be re-run in 1/2 hour
3. destroys the existing build.

If the file necessary.java is found:

1. sets the build environment variable $FILES to be the comma-separated list of files found in the directory. $FILES is prepended to the build environment so that other build environment variables may make use of it.

NOTE: line breaks are not right in this document. See also the javascript_step_example.txt file.

```
function step_main()  {
      setup();
      output_results();
}

function setup()  {
      var path = BF_Obj_JavaScriptStep.getBuildEnvironment("TESTPATH");
      if (!path) {
            BF_Obj_JavaScriptStep.terminateStep("No path directory
provided!", false);
      }

      var succeeded = BF_Obj_JavaScriptStep.executeCommand("cd " + path +
"\nls -l");
```

```
      if (!succeeded) {
            BF_Obj_JavaScriptStep.terminateStep("Problem with execution!",
false);
      }
}

function output_results()  {
      var found = false;
      var line = '';
      var filelist = '';
      while (line = BF_Obj_JavaScriptStep.getCommandResponseLine()) {
            var matches = false;
            if (matches = line.match(/[a-zA-Z_\-\.]*(java|txt)/)) {
                  BF_Obj_JavaScriptStep.printToLog("Found file '" + matches
[0] + "'");
                  filelist += matches[0] + ", ";
                  if (matches[0] == "necessary.java") {
                        found = true;
                  }
            }
      }
      if (!found) {
            BF_Obj_JavaScriptStep.printToLog("Necessary file not found!");
            BF_Obj_JavaScriptStep.failStep();
            BF_Obj_JavaScriptStep.setNotification(false, "Developer",
                        "Necessary file not found!", "The file
'necessary.java' was not found in the directory '" +
                        BF_Obj_JavaScriptStep.getBuildEnvironment("TESTPATH")
+ "' - the build will be requeued in 1/2 hour.")
            BF_Obj_JavaScriptStep.requeueBuild(1800);
      }  else {
            BF_Obj_JavaScriptStep.setBuildEnvironment("FILES", filelist,
                        ENVVAR_TYPE_SET, ENVVAR_PLACE_PREPEND);
      }
}
```