IBM Rational Build Forge Adaptors Help

For versions 7.0.1 and later

What is an Adaptor in Build Forge?

Spencer Murata

September 17, 2008

Topic **Page(s)**

Introduction

This document has been made available as a supplement to the information documented in *IBM Rational Build Forge Online Help*, and it provides some additional insight on what a Build Forge Adaptor is.

The Adaptor differentiates Build Forge from being just another build tool. It is made up of four components (Template, Interface, Command, and BOM Format), which are each detailed in this document. You will also learn how the Adaptor can be used to control branching logic for Build Forge.

Though the details in this white paper are applicable to Adaptors in general for any version of Build Forge, the supplied example is only applicable to version 7.0.1 and better.

Advisory: This document does not duplicate the information already published in *IBM Rational Build Forge Online Help*. If you have not already done so, it is advised that you first review that manual, which can help you gain a fuller appreciation of the subject covered in this white paper.

What is an Adaptor in Build Forge?

Just Another Build Tool

Build Forge is *not* a build tool. It is much more than that!

The traditional build tool:

1. Is restricted to a binary outcome.
2. The file will complete or it will not.
3. The make will succeed or it will not.

Build Forge is not limited to those outcomes, because it is a *process automation tool*.

Process Automation Tool

As a process automation tool, Build Forge must be able to handle many different logical paths from a single decision. Much like a real-life workflow, the path may not strictly be binary from a single point, and it can branch or loop any number of ways.

The question is how to represent this in Build Forge, and the answer is Adaptors!

Adaptors

Adaptors are the crucial bit of functionality that allows Build Forge to represent multiple branches from a single bit of input.

Without Adaptors, you are only using the regular steps in Build Forge which function much in the same way as a make file. The steps execute something on a client machine and get output back: *binary action*. That is the very basic kind of integration that is very easy to set up in Build Forge. But more than integration, it allows a rudimentary branching logic and looping ability.

XML Adaptor Example

This example XML Adaptor, named **ball.xml**, is used to control branching logic for Build Forge.

```
<!DOCTYPE PROJECT_INTERFACE SYSTEM "interface.dtd">
<PROJECT_INTERFACE IFTYPE="Test" INSTANCE="7.02">
<template>
      <env name="COLOR" value="Red-Green" />
      <env name="SIZE" value="Small" />
      <env name="BALL_SERVER" value="ball" />
</template>
<interface>
      <run command="determine_color" params="$COLOR"
server="$BALL_SERVER" dir="/" timeout="360" />
      <ontempenv name="Changed" state="empty">
            <step result="fail"/>
      </ontempenv>
<onproject result="fail">
      <notify group="BALL_WATCHERS" subject="Build failed!"
message="The $SIZE $COLOR ball failed!" />
</onproject>
</interface>
<command name="determine_color">
      <execute>
            echo $1
      </execute>
      <resultsblock>
            <match pattern="(.*?)-(.*?)">
                  <setenv group="ballcustomEnv" name="COLOR"
            value="$2"/>
                  <setenv name="Changed" value="TRUE" type="temp"/>
                  <bom category="ball_info" section="Ball Info">
                        <field name="Color:" value="$1-$2"/>
                        <field name="Date:" value="$CurDate"/>
                  </bom>
            </match>
      </resultsblock>
</command>
<bomformat category="ball_category" title="Ball Category">
      <section name="ball_section">
            <field order="1" name="color" title="Color"/>
            <field order="2" name="date" title="Current Date"/>
      </section>
</bomformat>
</PROJECT_INTERFACE>
```

Understanding the Adaptor Structure

Adaptor Structure

The Adaptor structure is broken into 4 components:

- Template
- Interface
- Command
- BOM Format

Template

The first section of the Adaptor is Template.

This is declared using the `<template>` tag. The template section is used to declare those variables that will be used by the Adaptor. If the Adaptor will need to use a particular variable declare it here.

Template Example:

```
<template>
     <env name="COLOR" value="Red-Green" />
     <env name="SIZE" value="Small" />
     <env name="BALL_SERVER" value="ball" />
</template>
```

In the *Template Example*, the Adaptor is declaring that it will be using COLOR, SIZE and BALL_SERVER for variables; hence, any project that uses this Adaptor will need to make sure that its environment includes those variables. The variables are only declared in the Adaptor template, but they are not created, by default.

The **Populate Env** checkbox uses the values in the template section to populate the environment. If you were to create an Adaptor link with a blank environment and the above Adaptor, the environment would be populated with variables COLOR, SIZE, and BALL_SERVER. The values would be Red-Green, Small and ball, respectively.

Interface

The second section of the Adaptor is the Interface.

The interface declares the entry point for Build Forge into the Adaptor. As of 7.0.1, there may only be one interface block. The interface block functionality remains the same though it defines the entry point for the Adaptor.

There are two major parts to the interface section: *commands to run* and *actions to take* depending on whether or not a step or project passes or fails.

Interface Example:
```
<interface>
     <run command="determine_color" params="$COLOR"
server="$BALL_SERVER" dir="/" timeout="360" />
     <ontempenv name="Changed" state="empty">
          <step result="fail"/>
     </ontempenv>
     <onproject result="fail">
          <notify group="BALL_WATCHERS" subject="Build
failed!" message="The $SIZE $COLOR ball failed!" />
     </onproject>
</interface>
```

In this *Interface Example*, we first run the command, **determine_color**, with the parameter $COLOR. It will run on server ball, in the root directory, and will timeout if there is no output in 6 minutes.

The commands that are used in the run command tags are declared in the command section. Run commands are essentially function calls. The tag maps to a command and will enter that command and execute it.

The **ontempenv** is the most important part of this Adaptor as it controls access to the step result tag. If the value of `Changed` is *empty* or `Changed` is *undeclared*, then the Adaptor step will fail! This means that the current build will not run; it will delete itself and remove any trace of itself.

This is the key functionality of using Adaptors, the build will not finish if the Adaptor does not pass. The last tag is setting up actions to take in the event that the project running our example Adaptor fails. In this case we are sending a notification to the group `BALL_WATCHERS` with a subject of *Build failed!* and the content is `The $SIZE $COLOR ball failed!`

Command

Third section of the Adaptor is Command.

This is where actions are declared and defined. This is where the bulk of the Adaptor will be defined.

The command component is split into two blocks:

1. There is the execute block, which defines what command line arguments you are looking to execute.

2. After that there is the results block, which takes the output from the execute block and parses it. The results block also takes the parsed data and puts it into variables.

Command Example:
```
<command name="determine_color">
     <execute>
     echo $1
     </execute>
     <resultsblock>
          <match pattern="(.*?)-(.*?)">
               <setenv group=ballcustomEnv name="COLOR"
value="$2"/>
               <setenv name="Changed" value="TRUE"
type="temp"/>
               <bom category="ball_info" section="Ball
Info">
                    <field name="Color:" value="$1-$2"/>
                    <field name="Date:" value="$CurDate"/>
               </bom>
          </match>
     </resultsblock>
</command>
```

This looks a little confusing, so let's simplify it.

The first part is the command tag. This is the declaration of the command. The command just names the command that we are going to define. In our example, we are creating a command named **determine_color**.

There is another parameter that we can use in the command tag that was not used in the example, but is worth mentioning. When talking about the name of the command being determine_color, it is helpful for us to add that we can also **set a mode** here. The syntax is `mode=?`, where ? can be either:

- `conjoined:` All calls to the command are grouped in one call for server processing.
- `parallel:` Calls are processed individually as server slots become available.
- `exec:` Commands are started and immediately processed by the server.

The next block is the execute block. It will call the contents of the execute block on the command line. We are assuming that echo is a valid shell command and will simply return the value of the parameter. In the example, we are also passing in a parameter used with the command call. Recall that when we used the command in our previous interface block we used `$COLOR` as a parameter. I will not go into the Perl of parameter passing, but suffice to say that $1 will return the first variable in an array, $2 the second, $3 the third and so on. The important thing to realize is that in this case $1 is $COLOR. So if we were to execute the equivalent command on the command line (assuming `$COLOR` is Red-Green) would appear as:

```
>echo Red-Green
```

Now we assume that our method is returning something on the command line. For this example, let’s assume the returned value is in the format:

```
>echo Red-Green
Red-Green
>
```

Now that we have our result, we need to parse it into meaningful bits for Build Forge. That is what the resultsblock block is for. The resultsblock block can have one or many match pattern blocks that will use a Perl regular expression to parse out the result of the execution block of the command. This regular expression has all the characteristics of a Perl regular expression and to understand all the syntax you need to refer to the appropriate third party documentation.

In our example, it is enough to know that the regular expression will look for a string of any length, then a hyphen “-” and then a string of any length after that. Due to the mechanics of Perl regular expressions that I will not go into here, the string before the hyphen “-“ will be stored in $1 and the string after will be stored in $2.

Now that we have our result parsed, we need to put it into the correct Build Forge variable. In the example, we are going to put it into the COLOR variable. To do this we are going to use the tag setenv. The setenv tag is versatile tag. In the above example it will take the result of the execution block and put it into the COLOR variable in the environment group ball_Adaptor. This will work the way that the .set dot command works.

The variable in the environment will be updated but will not affect the currently running build environment. So the next execution of our environment value for COLOR will be “RED”. Not very interesting, but in more complex examples the value would be different for the next execution.

Note: When you specify a group in setenv you **cannot** create a new variable, you **must** use an existing variable. Also you must use the form, [ADAPTOR], for the group name to make sure that whatever environment was associated with the Adaptor will be placed into the group automatically.

There are two other uses for setenv, if you do not have the group defined setenv will work like the .bset. The variable will be defined and declared if necessary for the scope of the currently running build, but will not affect the master environment. Meaning, if we changed the example above to read like:

```
<setenv name=“COLOR_RESULT” value=“$1”/>
```

… we will have declared a variable COLOR_RESULT whose value is not green. The variable would then be usable for the duration of the build.

The other use is to define a variable solely for use by the Adaptor, not the build. This is done by using **type** in the declaration. So if we look at the second setenv tag in the example:

```
<setenv name="Changed" value="TRUE" type="temp">
```

… we will have declared a variable **Changed** with a value **TRUE** for the duration of the Adaptor step. There are three values that type can be set to:

```
temp
temp append
temp once
```

The value `temp append` can be used to append new values to the end of the current value. Optionally, the characters after append will be inserted in between the old and new values. So if you declared:

```
<setenv name="Changed" value="TRUE" type="temp append\n">
```

… every time that has Changed was updated a carriage return would be placed in between the old and new values.

The `temp once` value is used to set a value immutably. Once the variable is declared with `temp once`, the value cannot be changed under any circumstance.

To recap the Command block section, it is important to realize that this is where the potential of Adaptors can be realized. We can easily think of the Command block as a function. It is reusable and has a built in conditional with the execute block being the conditional expression and the resultsblock block being the resulting action. This is where we can now define logical branches and recursive loops if so desired. Note that we can do much more than just integration here, we can logically branch our command flow!

BOM Tag

The last section of the example is the BOM tag.

BOM Tag Example:

```
<bom category="ball_category" section="ball_section">
     <field name="color" text="$1-$2"/>
     <field name="date" text="$CurDate"/>
</bom>
```

What is happening here is that we are defining custom entries for the **Bill of Materials** (BOM) for the build log to capture.

In the BOM tag we have defined a logical name for the BOM category and section. Also in the field tag we have defined a logical field name as well as the text associated with that field. At this point, it is important to remember that these are logical names that will be used in the next section.

BOM Format

The final part of the Adaptor is the BOM Format. This is where we organize how the BOM tags that we have previously declared will appear in the Build log.

BOM Format Example:

```
<bomformat category="ball_category" title="Ball Category">
     <section name="ball_section">
          <field order="1" name="color" title="Color"/>
          <field order="2" name="date" title="Current
Date"/>
     </section>
</bomformat>
```

Recall from the previous section we defined our logical name for the ball `BOM category` to be `ball_category`. So we use that logical name here for the category variable. Title indicates how the category will be displayed in the BOM. The same goes for the field tags. Notice that if you have more than one field associated with a section you will need to include the order variable to define the order that the fields are to appear in.

Summary

Adaptors are an important part to the functionality of Build Forge. They provide a degree of customization that allows Build Forge to control what builds will run.

The Build Forge Adaptor is an interface, made up of four major parts, that allows the system to interact with an external system. Our example demonstrates the use of an Adaptor to control branching logic for Build Forge.

Now you should have knowledge of the various parts of the Adaptor and how they fit together. The next step is to apply this knowledge to your environment.

Good luck with implementing your own Adaptors to fit your workflow needs!

References

Build Forge Online Documentation:
http://www.ibm.com/support/docview.wss?&rs=3099&uid=swg21251262

IBM Rational Build Forge Adaptor Toolkit:
http://www.ibm.com/software/awdtools/buildforge/toolkit/index.html