Getting Started with iRules: Delimiters

Jason Rahm, 2016-03-06

So far in this series we've covered introductions across the board for programming basics and concepts, F5 terminology and basic technology concepts, the core of what iRules are and why you'd use them, as well as cornerstone iRules concepts like events, control structures, and variables. In this article, we will add to the foundation with a detailed discussion on delimiters. Buckle up!

Whitespace

The primary (and oft overlooked) delimiter in tcl and iRules is whitespace.

In tcl, basically everything is a command. Commands have names, options, and arguments. Whitespace (space or tab) is used to separate the command name and its options & arguments, which are simply strings. A newline or semicolon is used to terminate a command.

In this example `log local0. "iRule, Do you?"` the components of the command when you break it down are:

Command	log local0. "iRule. Do you?"
Command Name	log
Command Parameter	local0.
Command Parameter	"iRule. Do you?"

The space character is also used to separate the elements in a tcl list.

Double Quotes

Let's take a closer look at that last example.

You might have noticed that the 2nd parameter above is a string consisting of multiple words separated by whitespace, but enclosed within double quotes. This is an example of grouping with double quotes.

Everything between a pair of double quotes " ", including newlines and semicolons, are grouped and interpreted as a single contiguous string. When the enclosed string is evaluated, the enclosing quotes are discarded and any whitespace padding is preserved. A literal double-quote character can be included in a string delimited by double quotes if it is escaped with a backslash (\" -- more on backslash substitution below). Variable substitutions are always performed within strings delimited by double quotes.

Avoid using double quotes on already-contiguous alpha-numeric strings because that results in an extra trip through the interpreter to create an unnecessary grouping. However, even in contiguous strings, if using any characters besides alpha-numeric it's a good idea to enclose the entire string in quotes to force interpretation as a contiguous string. For example, one of the most commonly used iRules commands, HTTP::redirect, takes a fully qualified URL as an argument which should be double quoted: `HTTP::redirect "http://host.domain.com/uri"`

Strings as operands in expressions must be surrounded by double quotes or braces to avoid being interpreted as mathematical functions.

Square Brackets

A nested command is delimited by square brackets [ ], which force in-line evaluation of the contents as a command. Everything between the brackets is evaluated as a command, and the interpreter rewrites the command in which it's nested by discarding the brackets and replacing everything between them with the result of the nested command. (If you've done any shell scripting, this is roughly equivalent to surrounding a command with backticks, although tcl allows multiple nesting.)

In this example, the commands delimited by square brackets are evaluated and substituted into the log command before it is finally executed:

```
log local0. "[IP::client_addr]:[TCP::client_addr] requested [HTTP::uri]"
```

Strings contained within square brackets which are not valid tcl or iRule commands will generate an "undefined procedure" error when the interpreter attempts to interpret the string inside the brackets as a command. The use of square brackets for purposes besides forcing command evaluation requires escaping or backslash substitution.

It's most optimal if you will be referring to the result of a command only once to use inline command substitution instead of saving the result of the command to a variable and substituting the variable.

Backslash Substitution (Escaping)

Backslash substitution is the practice of escaping characters that have special meaning ($ {} [] " newline etc) by preceding them with a backslash. This example generates "undefined procedure: 0-9" error because the string "0-9" inside the [ ] is interpreted as a command:

```
if { [HTTP::uri] matches_regex "[0-9]{2,3}"} {
TCL error: undefined procedure: 0-9     while executing "[0-9]
```

This example works as expected since the square brackets have been escaped:

```
if { $uri matches_regex "\[0-9\]{2,3}" } {
  body
}
```

This example also works since the contents of the square brackets within the braces are not evaluated as a command (and it's also a bit more readable):

```
if { $uri matches_regex {[0-9]{2,3}}}{
  body
}
```

(The 2 approaches above are roughly equivalent performance-wise.)

Another common use of backslash substitution is for line continuation: Continuing long commands on multiple lines. When a line ends with a backslash, the interpreter converts it and the newline following it and all whitespace at the beginning of the next line into a single space character, then evaluates the concatenated line as a single command:

```
log local0. "[IP::client_addr]:[TCP::client_addr] requested [HTTP::uri] at [clock \
  seconds]. Request headers were [HTTP::header names].  Method was [HTTP::method]"
```

Backslash substitution, when required does not substantially affect performance one way or the other.

Parentheses

The most common use of parentheses ( ) in iRules is to perform negative comparisons. It's a common mistake to exclude the parentheses and unintentionally negate the first operand rather than the result of the comparison, which can result in the mysterious 'can't use non-numeric string as operand of "!"' error: If the first operand is non-numeric, the evaluation will fail, since a non-numeric value cannot be negated in the mathematical sense.

Here's an example demonstrating the use of parentheses to properly perform a negative comparison. The following approach logs "no match", as expected, since $x is a string and not equal to 3:

```
set x xxx
if { !($x == 3) } {
  log "no match"
} else {
```

```
  log "match
}
```

However, this (erroneous) approach results in the runtime error mentioned above:

```
set x xxx
if { !$x == 3 } {
  log "no match"
} else {
  log "match"
}
TCL error: can't use non-numeric string as operand of "!"
while executing "if { !$x == 3 } {   log "no match" } else {   log "match" }"
```

Other uses for parentheses in iRules are for regular expression grouping of submatch expressions:

`regexp -inline (http://)(.*?)(\.)(.*?)(\.)(.*?)`

and to reference elements within an array:

`set array(element1) $val`

Braces

And last but by no means least, braces are the most extensively used and perhaps least understood form of grouping in tcl and iRules. They can be confusing because they are used in a number of different contexts, but basically braces define a space-delimited list of one or more elements, or a newline delimited list of one or more commands. I'll try to give examples of each of the different ways braces are commonly used in iRules.

Braces with Lists

Lists in general are space delimited words enclosed in braces. Lists, like braces, may be nested. This example shows a list of 3 elements, each of which consists of a list of 2 elements:

`{ { {ab} {cd} } { {ef} {gh} } { {ij} {kl} } }`

Braces with Subcommand Lists

Some commands use braces to delimit lists of subcommands. The "when" command is a great example: In every iRule, braces are used to delimit all iRules events, encapsulating the list of commands that comprise the logic for each triggered event. The first line of each event declaration must end with an open brace, and the last line with a closing brace, like this:

```
when HTTP_REQUEST {
  body
}
```

There are several other commands that use braces to group commands, such as switch:

Syntax:

```
switch condition body condition body
Examples:

switch {
  case1 { body }
  case2 { body }
}
```

or

```
switch {
```

```
  case1 {
    body
  }
  case2 {
   body
  }
}
```

Braces with Parameter Lists

Other commands use braces to define a parameter which is really a group of values - a list. The "string map" command, for example, requires a parameter containing the value pairs for substitution operations. The value pairs parameter is a list passed as a single parameter.

```
Syntax:

string map charMap string
Example ("a" will be replaced by "x", "b" by "y", and "c" by "z"):

string map {a x b y c z} aabbcc
```

Braces with Parameter Lists and Subcommand Lists Combined

And some commands use braces for both parameter lists and subcommand lists:

```
Syntax:

for start test next body
Example:

for {set  x 0 } { x = 10 } { incr x } {
  body...
  ...
}
```

Nesting Braces

Regardless of what context they are used in, multiple sets of braces can be nested. The innermost group will become an element of the next group out, and so on. All characters between the matching left and right brace are included in the group, including newlines, semicolons, and nested braces. The enclosing (i.e., outermost) braces are not included when the group is evaluated. There must be a matching close brace for every open brace -- even those enclosed in comments. Mismatched braces will generate a syntax error to that effect when you try to save the iRule. (Mismatches in other delimiters and invalid command syntax may also generate a mismatched braces error, so if you can't find mismatched braces, start commenting out lines without braces until the syntax check passes, then fix the problem in the last line you commented.)

Special Cases for Using Braces

You can also use braces to enclose strings, just as you would double quotes, but variables won't expand within them unless a command also inside the braces refers to them. Because of that difference, if you need to handle strings containing $, \, [], (), or other special characters, braces will work where double quotes won't.

Braces may also be used to delimit variable names which are embedded into other strings, or which include characters other than letters, digits, and the underscore, enforcing the start and end boundaries of the variable name rather than relying on whitespace to do so. This example results in a runtime error since the end of the first variable name is unclear:

```
set var1 111
set var2 222
```

```
log local0. "$var1xxx$var2"
TCL error: can't read "var1xxx": no such variable     while executing "log local0. "$var1xxx$var2""
```

But if you delimit the variable name in braces, it works:

```
set var1 111
set var2 222
log local0. "${var1}xxx$var2"

Returns:
111xxx222
```

Class names containing the dash character must be delimited by braces to ensure the class name is evaluated correctly. These are both valid class references:

```
matchclass $::MyClass contains "Deb"
matchclass ${::My-Class} contains "Deb"
```

Forcing Variable Expansion within Braces

If you want to use variables in the charMap value, you'll need to use a different way to pass a list to the string map command.

As I mentioned above, variables aren't automatically expanded inside of braces. However, for some commands that accept a list as a parameter, it makes sense to pass variables to the command within a list, which is delimited by braces. To pass a list of expanded variables and/or literal strings, you can use an embedded "list" command to echo both the values of the variables and the literals into the enclosing command.

This example shows how to pass a variable and a literal string as the charMap parameter to the "string map" command, substituting the string "NewName" wherever the string contained in $oldname is found in the HTTP Host header:

```
Syntax:

string map charMap string
Example:

string map [list $oldname NewName] [HTTP::host]
```

Other Resources

This article is by no means an exhaustive review of tcl delimiter use & optimization, but hopefully covers the basics well enough to get you started. Comments, corrections, & observations are welcome as always. Here are some additional resources you may want to check out:

- Summary of Tcl language syntax (tcl.tk)
- Summary of Tcl language syntax (sourceforge.net)
- Tcl Fundamentals
- Is white space significant in Tcl
- Quoting Hell
- Quoting Heaven!
- tcl list command

If you have detailed questions about this or any other iRules questions, please ask! Join us for the next article in this series when we will cover logging and comments.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com