iRules 101 - #11 - Events

Joe Pruitt, 2008-27-02

The cornerstone of an iRule is the event. An event is an iRule extension to TCL that enables modular based programming by allowing one to write code that executes only within the context of a certain state of a connection. This article will discuss some of the details around events and how they can be used within a single iRule as well as multiple iRules applied to the same Virtual Server. Other articles in the series:

Rule 1: A Virtual Server my have one or more iRules as included Resources.

It's important to start with the fact that an iRule is just a block of script code. It doesn't do much of anything outside of the context of a Virtual Server. iRules exist as "Resources" on a Virtual Server and a Virtual Server can have zero or more iRules in it's resource list. The benefits of applying more than one iRule to a Virtual Server will become apparent later in this article.

Rule 2: An iRule may consist of one or more different event blocks.

This should be self evident, from the many samples in the iRules CodeShare, that you do not need to have multiple iRules if you want to write code blocks for different events. One or more event blocks can be included within a single iRule. An example iRule that contains multiple events is as follows:

```
when HTTP_REQUEST {
  set uri [HTTP::uri]
  log local0. "Received Request for uri '$uri' from client.  Sending to server..."
}
when HTTP_RESPONSE {
  log local0. "Received Response for uri '$uri' from server.  Sending to client..."
}
```

Rule 3: An iRule may consist of one or more like event blocks.

It may not be as evident that you can include multiple "like" events within the same iRule. The catch here is that you must specify a priority for each "like" event to tell the iRules processing engine the order you wish to evaluate the event blocks. Each event block has a default priority of 500 and valid values for a priority are 0 to 1000 inclusive. See the documentation for the iRule "priority" command for more details on the usage of priority. The following iRule illustrates the use of multiple "like" events within the same iRule.

```
when HTTP_REQUEST priority 200 {
  log local0. "I'm a priority 200, I'll get executed second."
}
when HTTP_REQUEST priority 100 {
  log local0. "I'm a priority 100, I'll get executed first."
}
```

You may ask yourself: "Self, why would I want multiple 'like' events when I could just put all the code in a single event block and be done with it."

For trivial iRules, this isn't that apparent. But, when you start building a collection of iRules and get a little deeper into your use them, you'll likely start seeing patterns in your code. It's likely that all of your HTTP_REQUEST events start looking like this

```
when HTTP_REQUEST {
  log local0. "URI Requested [HTTP::uri]"
  # Do something unique.
}
```

A problem arises when you want to change that first line in all 100 of your iRules to something different. Maybe you are tired of seeing your logs filled up with uri requests, or maybe it's that you want to add something else to your input logging. You have two choices: First, you could modify each and every iRule with the same chunk of code and then change them all again next time you want to make modifications. Or, you could take a modular programming approach to the problem. How about moving that "common" section of code to it's own iRule and assign it a very low priority. Then, for each of your Virtual Servers that used to have one iRule attached to them you can add the "common" iRule as well as the custom code that is different for each Virtual Server (this is where the AH HA should be kicking in from above). The above iRule will now look like this

```
when HTTP_REQUEST priority 100 {
  log local0. "URI Requested [HTTP::uri]"
}
```

```
when HTTP_REQUEST priority 200 {
  # Do something unique
}
```

Now changing the logging data is modified in one iRule, not all 100 that you were previously managing. This is a very simplistic example, but consider the possibilities. If you are offloading authorization on your Virtual Servers, terminating SSL, performing HTTPS redirection, or many of the other fun things to do with iRules, you'll see the benefits of modularizing your code.

Rule 4: You can't call an iRule from another iRule

This question of calling an iRule from another iRule comes up quite often as well as how does one pass variables from one iRule to another. We have disabled the abililty to make "procedure calls" as well as the ability to directly call one iRule from another one. But with the use of multiple iRules and priority based "like" event blocks, it is fairly straightforward in configuring the order that event blocks are executed. Session variables can then be used to pass information from one event block to another. By default all variables are "connection" based (unless defined in RULE_INIT), meaning that they are present during the lifetime of the connection (request and response). After the connection terminates, the session variables are garbage collected.

In the first example above, you'll see that a variable containing the "[HTTP::uri]" was created in the first event and accessed in the second event. This can be done with differing as well as like events. The use of session variables can simulate the behavior of parameters in procedure calls.

Conclusion

The flexibility of enabling iRules to contain multiple different or alike events within the same iRule along with the ability to apply multiple iRules to the same Virtual Server facilitates the use of modular programming and enables iRule authors to efficiently design and maintain iRules.

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

F5 Networks, Inc.	F5 Networks	F5 Networks Ltd.	F5 Networks
Corporate Headquarters	Asia-Pacific	Europe/Middle-East/Africa	Japan K.K.
info@f5.com	apacinfo@f5.com	emeainfo@f5.com	f5j-info@f5.com