iHealth API Part 3 - A Little Code

jong, 2014-23-10

We finished the last article with exploring some of the data available in the API with a web browser that was helpful enough to render it in a mildly readable fashion. For most automation projects, however, we don't care if it's easy to read, as long as it's parseable and we can do something with the data that doesn't involve watching streams of xml scroll off the screen.

For today, we'll be working with the data from the diagnostics section of the API:

We'll use the same data from the API that we explored last time.

```
GET /qkview-analyzer/api/qkviews/0/diagnostics
```

For this script exercise, we'll be using a couple of tools working in the unix shell:

- bash 4.x
- xmlstarlet
- grep

When the diagnostics run against a QKView, they either are considered a 'hit' where the diagnostic found an issue in the QKView, or a 'miss', where the diagnostic ran, but found no issue. We'll develop a little script that will provide a quick summary of our hits and misses for a given QKView.

In the script we'll deal with the following steps:

- authentication
- diagnostics retrieval and asking for subsets of diagnostics
- diagnostics parsing and selective display

Let's start out by writing a couple of functions that will deal with the housekeeping that we need to do in order to connect to the API. First thing we have to do is authenticate ourselves with the credentials that we used when we built our iHealth account.

By authenticating, we will obtain a cookie in the HTTP response to our authentication request (if it's successful!) that we will send with every subsequent request to the API. This way, the API knows who we are, which qkviews in the system are ours, and knows that we've proven we're who we're claiming to be.

This is currently done by sending a form POST to the login server containing our credentials, and stashing the cookies we get back for later use.

curl provides exactly the sort of functionality that we need to perform API work, and bindings for the curl libraries are available for lots and lots of languages if you want to use something other than bash, or want to develop a bigger script later on.

We set up some initial values for later use:

```
13 readonly CURL=/usr/bin/curl
...
23 RESPONSE_FORMAT=${FORMAT:-"xml"}
...
32 ACCEPT_HEADER="-H'Accept: application/vnd.f5.ihealth.api+${RESPONSE_FORMAT}'"
```

Now that the housekeeping is done, we'll need an authentication function:

```
 72 function authenticate {
 73         user="$1"
 74         pass="$2"
 75         # Yup!  Security issues here! we're eval'ing with user input.  Don't put this code in
 76         CURL_CMD="${CURL} --data-urlencode 'userid=${user}' --data-urlencode 'passwd=${pass}'
 77         [[ $DEBUG ]] && echo ${CURL_CMD}
 78
 79         if [[ ! "$user" ]] || [[ ! "$pass" ]]; then
 80                 error "missing username or password"
 81         fi
 82         eval "$CURL_CMD"
 83         rc=$?
 84         if [[ $rc -ne 0 ]]; then
 85                 error "curl authentication request failed with exit code: ${rc}"
 86         fi
 87
 88         if ! \grep -e "sso_completed.*1$" ${COOKIEJAR} > /dev/null 2>&1; then
 89                 error "Authentication failed, check username and password"
 90         fi
 91         [[ $VERBOSE ]] && echo "Authentication successful" >&2
 92 }
```

The way this script is written, it uses the setting of environment variables on the commandline in order to protect the innocent. This allows us to pass sensitive information into a script without it needing to be embedded for others to discover, or show up in the output of ps on a shared machine.

You can hardcode it if you wish, but I'd recommend not doing so.

Once authentication succeeds, then we go out and grab the diagnostics for the qkview we specified as an argument to the script:

```
 94 function get_diagnostics_hits {
 95         qid="$1"
 96         CURL_CMD="${CURL} ${ACCEPT_HEADER} ${CURL_OPTS} https://ihealth-api.f5.com/qkview-ana
 97         [[ $DEBUG ]] && echo "${CURL_CMD}" >&2
 98         out="$(eval "${CURL_CMD}")"
 99         if [[ $? -ne 0 ]]; then
100                 error "Couldn't retrieve diagnostics for ${qid}"
101         fi
102         echo "$out"
103 }
```

Notice how we have set up an Accept header in the request? This tells the API what format you want the response to be in. Diagnostics have a special role in the API, so they are available in a couple format: xml, json, pdf, and csv. Everything else in the API is only available in xml or json. Use the Accept header to specify how you want your response data. If you don't use a response header, then the API assumes you want XML, and returns XML.

In this example we're asking for XML explicitly.

Now that we have a big pile of XML, we only want a couple pieces of it, so make xmlstarlet dig through it and give us a nice display:

```
131         #perform some XML extraction, and print it out in a nice readable format
132         diagnostics_count=$(echo ${diagnostics} | ${XMLPROCESSOR} select -t -c 'string(/diagn
133         for ((i=1;i<=diagnostics_count;i++)); do
134                 printf "%-10s : " $(echo ${diagnostics} | ${XMLPROCESSOR} select -t -c "strin
135                 printf "%s\n" "$(echo ${diagnostics} | ${XMLPROCESSOR} select -t -c "//diagno
136         done
```

See how easy that is?

We can do the same thing with json if we want:

```
140         # in json
141         diagnostics_count=$(echo ${diagnostics} | ${JSONPROCESSOR} -r .diagnostics.hit_count)
142         for ((i=0;i<=diagnostics_count;i++)); do
143                 printf "%-10s : " $(echo ${diagnostics} | ${JSONPROCESSOR} -r .diagnostics.di
144                 printf "%s\n" "$(echo ${diagnostics} | ${JSONPROCESSOR} -r .diagnostics.diagn
145         done
```

If you need to deal with json at the commandline, and don't already know about it, jq is an incredibly powerful tool for handling json, and well worth your time exploring (see Working with iControl REST Data on the Command LIne for more details.)

So now that it works, what do all the fields mean? Let's look at the json output and see what we're getting:

```
{
  "diagnostics": {
    "filter": "hit",
    "diagnostic": [
      {
        "name": "H371501",
        "output": [],
        "run_data": {
          "h_importance": "MEDIUM",
          "match": true
        },
        "results": {
          "h_action": "Upgrade to version 10.2.2 or later.",
          "h_name": "H371501",
          "h_header": "A known issue causes the chmand process to leak memory on this BIG-IP platform
```

```
          "h_summary": "Due to a known issue, the chassis manager daemon, chmand, leaks memory on BIG
          "h_sols": [
            "http://support.f5.com/kb/en-us/solutions/public/12000/900/sol12941.html"
          ]
        }
      },
```

- name - the diagnostic name
- output - is an array containing any qkview specific information the diagnostics wants to tell you about
- run-data - data about the run including
    - importance (low medium high critical)
    - match (if it's a hit or not)
- results
    - h_action - references to more materials
    - h_name - the diagnostic name again
    - h_header - a title
    - h_summary - a more verbose explanation of the issue
    - h_sols - links to solution articles that go into more depth

Customizing the data that is shown in your summary is very easy, by adding the statements that dig into the diagnostics return data. One could, if one wished, build up a reporting system that showed the change in diagnostics results over time for a collection of qkviews if one were so inclined.

The full script is available here.

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