Using JSON in a Database Oriented Application on the JNIOR Series 4 Controller

Bruce S. Cloutier
INTEG Process Group, Inc., bruce.cloutier@integpg.com

***Abstract* – The JNIOR Controller is a generic device that is used in a variety of control and monitoring solutions. Application programs are executed by the operating system and can be used to optimize the controller for any particular purpose. Applications that collect data often can benefit from the services of a database engine. The JANOS operating system on the Series 4 JNIOR controller supports native JSON functionality. This paper demonstrates how JSON can provide simple database functionality in the absence of an actual database engine. This has the added benefit of creating database files that are directly compatible with dynamic web pages.**

Index Terms – JSON Database, Database Engine, Monitoring, JNIOR Configuration, Honeypot, Mirai Malware.

JNIOR CONTROLLER

The JNIOR is a networked controller which can be considered to be a member of the *Internet of Things* (IoT). It was developed long before these terms and classifications were coined. It was intended to be, and successfully achieves the role of an inexpensive alternative to more elaborate and costly industrial control systems. This format of device has been available in a series of successive compatible models for over 15 years. It provides integrators with a consistent and reliable source of controller hardware.

The term JNIOR is an acronym which stands for Java Network Input Output Resource [1]. It can be pronounced as "junior" although the name is not intended to diminish the device's role in the world of controllers. In fact, beyond simply providing remote I/O resources, the JNIOR is programmable and is capable of providing sophisticated distributed control logic and **data monitoring**. It can perform autonomously as a stand-alone system.

JANOS OPERATING SYSTEM

The JNIOR comes with its own *Operating System*. This is a complete preemptive multitasking operating system with all of the generic functionality found in multiuser computing systems. This includes a full network stack with secure protocols and a fully functional web server with server-side scripting capabilities.

The JANOS Operating System has been completely developed by INTEG and specifically for use with the JNIOR. This has allowed the system to be optimized for its function without extraneous overhead. It allows the inexpensive microcontroller at the heart of the JNIOR to perform the tasks of more complicated computing platforms.

Being fully developed by INTEG this OS contains no third-party source code. The advantage here is that when issues are identified, INTEG is 100% in a position to resolve the problem. In fact, many issues can be corrected the same day that they are identified. There is no one else to blame and issues cannot be "elevated". There is no outside group wherein issues become subject to separate prioritization, independent release schedules, and code obsolescence. Issues just get resolved.

Perhaps as important is the ability to easily expand functionality. For example JANOS natively implements JavaScript Object Notation (JSON) [2] in direct support of external web interfaces. This feature can have other benefits. We will elaborate in this paper through an example demonstrating how this JSON functionality can be used to provide database support for application programs.

APPLICATION PROGRAMMING

The JNIOR out of the box is fully functional as a remote I/O device. It can be used without running any specialized *Application Program*. An application program however gives the user the ability to add unique and custom capabilities to the device. INTEG provides stock application programs such as *cinema.jar* which creates an environment specific to Digital Cinema where cues and other signals trigger definable macros. While there is a set of available programs, additional programs can be developed by the user to support a wide range of custom needs.

You can do almost anything with an application program. Providing documentation to support that flexibility is somewhat difficult. INTEG is therefore willing to freely support your programming efforts. All you have to do is contact us and let us work with you.

Since the introduction of the Series 4 JNIOR, application programming has been greatly simplified. These programs are written in standard Java and any of the available compiler tools can be used. You need only generate a JAR file. We are most familiar with *Netbeans* where there are simple configuration steps that insure that your project is built against the *JanosClasses.jar* runtime environment. Your program needs to be compiled to run under this specific runtime and to not reference any standard

classes as might be supplied for programs targeted to run on PCs and other devices.

The *JanosClasses.jar* runtime file can be retrieved from the JNIOR as it resides in the */etc* folder. This library is JANOS version specific and programs compiled against one version are guaranteed to be compatible with later versions. A version of this JAR file is available that contains Javadoc and source code references in addition to the set of runtime classes.

HONEYPOT TESTING

The JNIOR is a secure device. It is expected to be completely functional and secure in difficult environments such as the Internet. History has shown us that this open network environment is hostile and devices such and the JNIOR can fall victim to malicious attacks. INTEG has placed a development unit on the open network at the address *honeypot.integpg.com*. We monitor its performance daily.

When a controller such as the JNIOR is placed in a potentially hostile network environment, we recommend that the unit be configured to minimize the network footprint. Only necessary functions (ports) should be left open/active. Certainly a minimum number of user accounts should be active and the passwords should be strong (especially not left as default passwords). Access should be through secure protocols such as HTTPS.

Immediately at the start of these honeypot tests we detected a high level of illicit login attempts on the unit's Telnet port (23). Analysis proved that the Mirai Botnet [3] was the leading source of this activity. As JANOS logged the login failures and the quantity of reports grew, it occurred to us that an application to track and map the source locations for each attack would be interesting to develop. Where in the world are all of these infected machines? This also would give the HoneyPot JNIOR something interesting to do while expanding the scope of our testing.

JNIOR LOG FILES

The JNIOR creates a number of log files. Log files typically grow with time and can get quite large. Embedded devices often have limited memory resources and it is necessary to limit the size of log files. One strategy used by the JNIOR allows a log file to grow to about 64KB at which point the file becomes a BAK file and a new log begins to grow. Any previous BAK file is overwritten in the process. So at any one point there can be from 64KB up to 128KB of log detail available.

The Mirai login failure reports would quickly overwhelm the standard *jniorsys.log* system log. As this is a valuable source of message detail when issues do arise, it became necessary to move the login failure reports to a separate *access.log* file. This change preserves the other system log entries for a much longer period of time improving our ability to debug in the future.

THE TRACKING APPLICATION

The approach we used in this application was to wait for a new version of the *access.log.bak* file to become available. At that point we process the entire content and return to wait for another updated BAK file. Scanning for new entries in the *access.log* file and trying to provide a more real-time representation is problematic. There would be contention between the application and the system as content is processed at the same time new entries are being added. The JNIOR mediates that but there is no reason to stress the system to that point. The BAK file updates often enough given the current status of this botnet.

Once a remote computer infected with the Mirai malware finds an active Telnet port on a random system it attempts to login. The Mirai source code has been made openly available and we can see that it uses a predefined list of account and password combinations. It is looking for specific targets hoping that users fail to change login credentials from their defaults (or otherwise use common passwords). This means that for any one botnet host there are numerous login attempts and hopefully failures. On the JNIOR each failure generates an entry in the *access.log* file.

We want to track each host. Therefore we need to create a database of host information and one that is keyed on IP Address. This is accomplished using the native JSON functionality. You will see that we can age/remove old host entries from the database, we can add new entries to the database, and the database can be used directly by a web page designed to map the set of infected hosts.

The main structure of the resulting application is shown in Figure 1. The JAR file has been loaded into the */flash* folder on the unit and we created a Run key in the Registry to start the program on boot.

WEBSERVER SUPPORT

The application program will detect a new set of login failures and process the entire list. The database is updated in the process and then saved as */flash/public/infected.json*. We will look at that process shortly. Here we see that the database is made publicly available by its location.

The JANOS WebServer is a fully functional server capable of handling multiple simultaneous requests. This includes a version of server-side scripting modelled after PHP [4]. The website root where web pages are stored is */flash/www* and by default requires login authentication. This is important especially as we are running in a hostile network environment. The WebServer supports default pages such as the Dynamic Configuration Pages (DCP) used to remotely configure and control the JNIOR. This absolutely needs to be password protected.

JANOS also provides an additional root area wherein files are automatically made public. This is the folder */flash/public*. So while we are able to protect everything on the server, we are also able to provide open access to selected files simply by placing them in this special folder. In this application that includes the JSON database

```
1   public class JAccess {
2
3     public static void main(String[] args) throws Throwable {
4
5       // Program is restarted via watchdog should any exception be thrown. See error.log
6       //  for details.
7       Watchdog watchdog = new Watchdog("Access logger");
8       watchdog.setAction(Watchdog.WDT_RESTART);
9       watchdog.activate(300000);
10
11      // background routine updates infected.json file from access.log.bak periodically
12      for (;;) {
13
14        // check file timestamps
15        File jdb = new File("/flash/public/infected.json");
16        File acc = new File("/access.log.bak");
17
18        // if the access log has changed
19        if (acc.lastModified() >= jdb.lastModified()) {
20
21          // database
22          Json db = new Json(jdb);
23
                .
                . (see Figures 2 and 5 for the code here)
                .

95          db.save(jdb);
96        }
97
98        // sleeps alot
99        Thread.sleep(60000);
100       watchdog.refresh();
101     }
102   }
103 }
```

FIGURE 1
ACCESS TRACKING APPLICATION – MAIN STRUCTURE

that we have created along with a *map.php* webpage that displays the content. The latter utilizes JavaScript and retrieves the database though an AJAX [5] call. Since this database is already formatted as JSON it can be easily used in this case to generate the map. This page is available as *honeypot.integpg.com/map.php* and no login is needed.

FAULT TOLERANT SUPPORT

Applications can encounter errors. This is especially true in a situation like this where an external service is used to obtain valid Latitude and Longitude estimates for an IP address' location on the globe. Communications with the service may fail due to no reason of our own. For a Java application this means that an exception would be thrown.

Typically the developer in a finished application would use a *try-catch* construction to trap any exception and to then proceed appropriately. Another approach that is often useful during development and when the types and sources of possible exceptions are not yet known is to let the program throw uncaught exceptions. The compiler allows this when the *throws Throwable* clause is present as it is on Line 3 in the program. When an uncaught exception occurs it is reported to the *errors.log* file and the program terminates.

A program termination obviously interrupts operation and is generally not acceptable performance. Here we show the use of the *Watchdog* class. Lines 7-9 define a watchdog for the program which will restart program if the watchdog's timer ever expires. In this case a fairly tolerant timeout of 5 minutes is used. We see that on Line 100 the timer is reset periodically as part of the main program loop and we are good so long as the program is running. The watchdog is a system level service.

By using the *Watchdog* class we insure that our program is always running. This approach then allows us to collect information about errors in the *errors.log* file. This information would later be useful in refining/debugging the application.

MAIN PROGRAM LOOP

The main program loop (Lines 12 thru 101) runs forever. Each iteration of the loop ends up sleeping for 60 seconds (Line 99) and servicing the watchdog timer (Line 100). The overall task here is to detect the change in the *access.log.bak* file. This is achieved by comparing the last modification timestamp of the access log file to that on the database that we will create and update. If the access log has been updated

```
24          // age database content
25          long oldage = acc.lastModified()/1000 - 24*60*60;
26
27          String[] keys = db.keyarray();
28          for (int n = 0; n < keys.length; n++) {
29            Json data = (Json) db.get(keys[n]);
30
31            long timestamp = data.getLong("timestamp");
32            if (timestamp < oldage)
33              db.remove(keys[n]);
34          }
35
```

FIGURE 2
AGING – REMOVAL OF OLD ENTRIES

since we last updated the database then we have new data to process. Lines 15 and 16 reference the files and Line 19 performs the check.

When the access log backup is ready to be processed we open the database and begin the work. Here we encounter our first use of a JSON object. Line 22 opens the JSON data file and loads the structure into memory where we can access and modify previously collected data. Note that this creates an empty database if the JSON file does not yet exist.

Once the database is open we can proceed to process the data. When we are done the database is updated/saved by Line 95. The program then gets to go back to sleep and waits for the next *access.log.bak* update.

DATABASE AGING

At this point we need to look at the resulting JSON structure for the entries that we eventually add to the database. Figure 3 shows a typical entry in the array. Our JSON database amounts to one large array of objects each identified uniquely by a textual representation of the host's IP address.

```
"115.20.188.5":{
  "as":"AS4766 Korea Telecom",
  "city":"Jincheon",
  "country":"Republic of Korea",
  "countryCode":"KR",
  "isp":"Korea Telecom",
  "lat":36.5743,
  "lon":128.4943,
  "org":"Korea Telecom",
  "query":"115.20.188.5",
  "region":"47",
  "regionName":"Gyeongsangbuk-do",
  "status":"success",
  "timezone":"Asia/Seoul",
  "zip":"",
  "timestamp":1501547484
}
```

FIGURE 3
EXAMPLE JSON ENTRY

This JSON object contains a number of fields most of which are returned by a third-party service whose response to our query is also in JSON. Of interest is the last field which the application program inserts. This *timestamp* specifies the second at which the entry was added to the database. Since our application wishes to track only the login attempts that occur over the past 24 hours we need this field.

Figure 2 shows the program steps that we first perform after opening the database. This is our first real database oriented procedure. Here we want to scan the entire database and remove entries that are older than 24 hours. Line 25 calculates that point in time 24 hours prior to now. We use the last modification timestamp on the *access.log.bak* file although the application could easily use the current time.

Line 27 defines a String array containing each of the IP addresses referenced in the database. Those are our keys into the database. Lines 28 thru 34 then process each IP address by first obtaining the object (Line 29), retrieving the timestamp field (Line 31) and checking its age (Line 32). If the entry is found to have expired it is simply removed from the database with the statement on Line 33.

Once this step has completed our database contains only information about hosts that were encountered in the past 24 hours. Here we see how the JSON structure allows us to uniquely index the entries by IP address. We can then retrieve data by IP address.

SCANNING THE ACCESS LOG

The next step is to process the new entries that are now contained within the *access.log.bak* file. Figure 5 shows the next block of code that deals with the access log content. Here we open the log file (Lines 36 and 37) and read each line that file contains in a loop. Each entry contains a failed login report formatted as shown in Figure 4.

```
08/01/17 16:08:33.673,
   ** Command/187.162.255.236:41695 failed CMD
   login #1 'user' (pid 39450)
```

FIGURE 4
TYPICAL FAILED LOGIN REPORT

With each access log entry we apply a Regex [6] expression locating the IP address in the text. If one is found we attempt to read any prior reference to it in the database. If the entry already exists then there is no need to process it

```
36          FileReader fin = new FileReader("/access.log.bak");
37          BufferedReader in = new BufferedReader(fin);
38
39          String line;
40          while ((line = in.readLine()) != null) {
41
42            Pattern p = Pattern.compile("\\d+\\.\\d+.\\d+.\\d+");
43            Matcher m = p.matcher(line);
44            if (m.find()) {
45
46              String ipaddr = m.group();
47              if (db.get(ipaddr) == null) {
48
                    .
                    . (see Figure 6 for the code here)
                    .

90              }
91            }
92          }
93
94          in.close();
```

FIGURE 5
SCANNING THE FAILED LOGIN ENTRIES – ACCESS.LOG.BAK

further. We are only interested in discovering new addresses that we will then need to add to the database.

Once we have made our way through the entire access log we must close the file (Line 94). At this point we are done and we fall back into the main loop of Figure 1. The database is saved and the program proceeds to sleep.

PROCESSING A NEW HOST

Figure 6 show the balance of the program filling in Lines 49 thru 89. Here we perform the interaction with the external service that will supply an approximate Latitude and Longitude for the IP address of a new infected host. We will add that information into the database and continue processing access log entries.

The procedure shown in Figure 6 creates a connection to the external service and issues a request for any information available regarding the specific host IP address. The details here are beyond the scope of this paper. There are numerous services of this kind. Each provides data in differing formats. Here we use a free service that also optionally replies using the JSON format.

As a point of interest since we are dealing with an international landscape we need to handle the expanded Unicode [7] character set. In this case these characters are encoded using UTF8 [8]. On Line 79 we handle the UTF8 conversion. Note that JANOS supports these features in a manner that closely resembles standard Java in order to maintain a familiar programming environment.

Finally Line 80 verifies the JSON format and Lines 81 thru 83 create and add the record to the JSON database. An object is directly created from the textual JSON response supplied by the service. This is why our entries contain all of the various fields supplied by the service. We append the *timestamp* field here using the system time in seconds. This is what we need to age the entries removing those that then become older than 24 hours. The new record is added to the database and we proceed to process any additional failed login reports.

WEB INTERFACE

With the application running the */flash/public/infected.json* database file updates periodically as Mirai login attempts go on continuously. At the time of this writing this occurs every 3 hours or so. The *honeypot.integpg.com/map.php* page uses the Google mapping API [9] to display a simple map with markers. Those markers are derived from our database file directly using JavaScript.

Once the page loads a small piece of JavaScript runs and retrieves the database using an *XMLHttpRequest()*. The JNIOR responds with the database file content which is textual JSON. This is easily parsed by JavaScript into an object that can be as easily processed as it was created.

The web page's script then scans through the database adding a marker for each IP address using the 'lat' and 'lon' fields. Here we format tooltip mouseover information using the 'ipaddr', 'city', 'country' and 'regionName' fields. These are logically combined and neatly formatted with abbreviations to produce a nice location reference.

SUMMARY

This is an interesting application for the JNIOR that demonstrates its potential value outside of the realm of controls. The ability of the device to monitor various signals of a hardware nature and of a network nature, offers the capacity for data collection. That data can be easily transmitted to remote servers or added to a local database. These metrics can be of significant value in some markets.

We have demonstrated the use of the JSON structure in a database application. Our application needed to store

```
49              String serverHostname = "ip-api.com";
50              int port = 80;
51
52              Socket dataSocket = new Socket(serverHostname, port);
53              PrintWriter sockout = new PrintWriter(dataSocket.getOutputStream(), true);
54              BufferedReader sockin =
55                new BufferedReader(new InputStreamReader(dataSocket.getInputStream()));
56
57              sockout.println("GET /json/" + ipaddr + " HTTP/1.1");
58              sockout.println("Host: " + serverHostname);
59              sockout.println();
60
61              // obtain data length from header
62              int length = 0;
63              String response;
64              while ((response = sockin.readLine()) != null) {
65                if (response.length() == 0)
66                  break;
67                if (response.startsWith("Content-Length: "))
68                  length = Integer.parseInt(response.substring(16));
69              }
70
71              if (length > 2) {
72                char[] resp = new char[length];
73
74                int count = 0;
75                while (length - count > 0)
76                  count += sockin.read(resp, count, length - count);
77
78                response = new String(resp);
79                response = new String(response.getBytes(), "UTF8");
80                if (response.startsWith("{") && response.endsWith("}")) {
81                  Json data = new Json(response);
82                  data.put("timestamp", System.currentTimeMillis()/1000);
83                  db.put(ipaddr, data);
84                }
85              }
86
87              sockout.close();
88              sockin.close();
89              dataSocket.close();
```

FIGURE 6
GETTING HOST DETAIL – UPDATING THE DATABASE

information about individual host computers which can later be retrieved based upon IP address. The native JSON support provided by JANOS offers that functionality. Not only could we easily store information but were able to age and remove old records. The database allowed us to determine if a host was new to us. And, as a bonus the resulting database file is directly usable in rendering web page content.

Nevertheless this is not a substitute for an actual database engine. Here JSON objects are constructed of and stored as textual strings. The string representations of numbers are not in any way as efficient as binary formats. Arrays are searched linearly where a database engine would handle indexing using more advanced techniques aimed at minimizing access timing. The functions of a database engine are not yet available in JANOS. It is clear though that for many simple applications the JSON approach is a workable option.

On the application programming side this example utilizes some complicated techniques. The code here has not been optimized. Some programmers will likely notice right away where gains in efficiency can be achieved. For example we repeatedly regenerate the same Regex object. That can certainly be done once during initialization. The database saves all of the data returned by the IP location service. Clearly we could save only what we need. That would reduce the database size, lower access times, and even reduce the page load timing for browsers. Even in its current form the application runs as expected and handles its tasks quickly.

One final note, the honeypot testing that has been ongoing has greatly improved JANOS and hardened the product in regards to network security. While it is likely that any device specifically targeted by professional hackers would eventually fall, the JNIOR with JANOS (v1.6.2) or later quietly performs admirably as an anonymous member of the IoT.

REFERENCES

[1] Java™ is a trademark of the Oracle Corporation and its related entities.

[2] JSON JavaScript Object Notation is a lightweight data-interchange format, www.json.org

[3] Mirai Botnet (malware) - turns networked devices running Linux into remotely controlled "bots" that can be used as part of a botnet in large-scale network attacks, Wikipedia

[4] PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used open source general-purpose scripting language that is especially suited for web development and can be embedded into HTML, www.php.net

[5] AJAX - Asynchronous JavaScript And XML, allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes, www.w3schools.com/xml/ajax_intro.asp

[6] Regex, Regular Expression, a pattern used in string searching algorithms, Wikipedia, en.wikipedia.org/wiki/Regular_expression

[7] Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems, Wikipedia, Unicode.org

[8] UTF8 one way of encoding Unicode characters, Wikipedia, en.wikipedia.org/wiki/UTF-8

[9] Google Maps APIs, developers.google.com/maps/

CONTACT INFORMATION

INTEG Process Group, Inc.,
2919 E Hardies Rd 1st Floor
Gibsonia, PA 15044
USA

724-933-9350
www.integpg.com