1 Introduction

Two months ago I was complaining that I didn't have a job, and then suddenly I had one[1]. I work for Splunk[2] now, an enterprise-level machine data analysis company. Their core product is pretty magical: you can shove just about any logfile, real-time stream, script-generated datasource, or anything else at it, and then interrogate the data to understand and monitor your networks, server farms, client connections, whatever. It does an amazing job of correlating data through little more than keyword observation. There's even a free version[3], which, while limited to a half-gig of data, is a good way to start off taking apart your weblog files.

They have me niftying up the third-party web framework, the thing large companies use to integrate the data we collect with their own network intelligence dashboards, visualization systems, whatever. It's all my kind of thing: Python back-end, Javascript/Backbone/jQuery front-end stuff, lots of clever closures and event handling.

But I decided, for entertainment purposes only, to learn more about a part of the system I know almost nothing about: the Java SDK. And it wasn't enough to work in a language I don't know (since I don't know Java) with SDK's I've never seen before. I had to go make it work with a language that, to the best of my knowledge, no one has ever demonstrated compatibility with Splunk before. I had to make it work in Clojure[4].

Yes, I'm a Hipster Hacker. I mean, come on, if you're not making it difficult for you, what is education for, anyway?

1.1 Disclaimer

I am a newbie to all of this. Clojure, Splunk, even Java. This is entirely raw and beginner level stuff. I did this for my own edification. This code in no way represents the state of the art at Splunk. It's definitely not warranted in any way by me or anyone else, and it's licensed under the Apache Public License, V2. Use at your own risk. Don't make me go ALL CAPS on you.

1.2 Literate Program

A note: this article was written with the Literate Programming[5] toolkit Noweb[6]. Where you see something that looks like «this », it's a placeholder for code described elsewhere in the document. Placeholders with an equal sign at the end of them indicate the place where that code is defined. The link (U->) indicates that the code you're seeing is used later in the document, and (<-U) indicates it was used earlier but is being defined here.

[1] See URL `http://www.elfsternberg.com/2013/03/14/gave-lightning-talk/`.
[2] See URL `http://splunk.com`.
[3] See URL `http://www.splunk.com/download`.
[4] See URL `http://clojure.org/`.
[5] See URL `http://en.wikipedia.org/wiki/Literate_programming`.
[6] See URL `http://www.cs.tufts.edu/~nr/noweb/`.

1.3 Install everything

First, install Clojure, Leiningen, the Splunk free server, and the Splunk Java SDK[7]. Start up the free server and give it some data. I gave it the dataset from earthquake.usgs.gov– "earthquakes greater than Richter 2.5 in magnitude over the past 30 days.") but you can give it whatever you want.

Completing the installation is a bit tricky because this example relies on both the Splunk Java SDK, which is not in any Clojure or Maven repos, and the Splunk Java SDK's Command utility class, which Splunk has provided as a helpful tool for unpacking the command line and for understanding the Splunk command file (.splunkrc on most Linux and Mac boxes). You'll have to download the Splunk Java SDK yourself and install the Splunk JAR file. You'll also have to create, from the root of this project directory, resources/com/splunk, and deposit Command.class (which can be found in the SDK's tree somewhere) in the newly created directory.

I used lein localrepo to install the Splunk JAR file. It seems to have worked for me.

1.4 Create the project

In your workspace directory, whatever you call it, build a new project and call it `splunk`.

```
lein new splunk
```

Open the new directory and create a new directory, `lib`. Find the Splunk jar file in the SDK, and copy it into `lib`. Edit project.clj so it looks something like this:

2 ⟨*project cli rev 1* 2⟩≡

```
(defproject splunksearch "0.1.0"
  :description "SplunkSearch: An implementation of the Splunk Search example program in Cl
  :url "https://github.com/elfsternberg/splunk-search"
  :license {:name "Apache Public Licence 2.0"
            :url "http://www.apache.org/licenses/LICENSE-2.0"}
  :dependencies [
                 [org.clojure/clojure "1.4.0"]
                 [commons-cli "1.2"]
                 [gson "2.1"]
                 [opencsv "2.3"]
  :plugins [[lein-localrepo "0.4.1"]]
  :main splunksearch.core)
```

[7] See URL `http://dev.splunk.com/view/splunk-sdk-java/SP-CAAAECN`.

Run `lein deps` to install your dependencies.

Then, use the `localrepo` command to install the splunk jar in your Maven repository:

```
lein localrepo coords splunk-1.1.jar | xargs lein localrepo install
```

Now, edit the project file to show your new dependency:

3a ⟨*project cli rev 2* 3a⟩≡

```
(defproject splunksearch "0.1.0"
  :description "SplunkSearch: An implementation of the Splunk Search example program in Cl
  :url "https://github.com/elfsternberg/splunk-search"
  :license {:name "Apache Public Licence 2.0"
            :url "http://www.apache.org/licenses/LICENSE-2.0"}
  :dependencies [
                 [org.clojure/clojure "1.4.0"]
                 [commons-cli "1.2"]
                 [gson "2.1"]
                 [opencsv "2.3"]
                 [splunk "1.1"]]
  :plugins [[lein-localrepo "0.4.1"]]
  :main splunksearch.core)
```

Now you're ready to begin. Fun, huh?

1.5 Create a simple query

The Splunk Java SDK has its own custom methods for dealing with arguments from the command line, from your run commands file (all those files in your home directory that end in "rc" on Linux and sometims Mac OS), and arguments to send to the remote Splunk server. We need two sets of arguments, one to define access to the service, and one to define the output mode of the data we expect to get back.

First, we have to create the namespace and import everything we're going to need:

3b ⟨*create the namespace* 3b⟩≡ (9a)

```
(ns splunksearch.core
  (:require [clojure.java.io :refer :all])
  (:import (com.splunk Service ServiceArgs Args ResultsReaderJson ResultsReaderCsv Results
  (:import (com.splunk Command))
  (:import (java.io InputStreamReader OutputStreamWriter)))
```

I pretty much followed the code in order, a list of procedures. Remember, this document serves mostly as a stream-of-consciousness "this is what I learned about Clojure while wrestling with Splunk" history of my project.

The Splunk `Command` class requires a series of definitions added, and then it attempts to parse the command line. Command instantiates via a static factory method, so that's accessed with a slash; the `doto` macro allows me to access the created object repeatedly, passing methods and arguments, and guarantees the object created is returned regardless of the return object of the last method. Clojure args have to be coerced via `into-array` into a Java array for Java-based args parsers to make sense of them.

4 ⟨*parse command arguments* 4⟩≡ (9a)

```
(defn build-splunk-command [args]
  (let [command
        (doto (Command/splunk "search")
          (.addRule "count" Integer
                    "The Maximum Number of results to return (default: 100)")
          (.addRule "earliest_time" String
                    "Search earliest time")
          (.addRule "field_list" String
                    "A comma-separated list of the fields to return")
          (.addRule "latest_time" String
                    "Search latest time")
          (.addRule "offset" Integer
                    "The first result (inclusive) from which to begin returning data. (def
          (.addRule "output" String
                    "Which search results to output {events, results, preview, searchlog,
          (.addRule "output_mode" String
                    "Search output format {csv, raw, json, xml} (default: xml)")
          (.addRule "reader"
                    "Use ResultsReader")
          (.addRule "status_buckets" Integer
                    "Number of status buckets to use for search (default: 0)")
          (.addRule "verbose"
                    "Display search progress")
          (.parse (into-array String args)))]
    (if (not= (count (.args command)) 1)
      (Command/error "Search expression required" nil))
    command))
```

My example code was taken from the `Search/Program.java` file, provided with the SDK. That program had a ton of local variables to control search generation, stream configuration, reader and output generation. I decided that all had to go into a simple map, which I could then refer to at any time.

Yes, the names above are repeated here. That's a bit of a code smell, I think.

5a ⟨*build the arguments map* 5a⟩≡

```
(defn build-argument-map [command]
  (let [opts (.opts command)
        ruleset [["count" 100]
                 ["earliest_time" nil ]
                 ["reader" false]
                 ["verbose" false]
                 ["field_list" nil ]
                 ["latest_time" nil ]
                 ["offset" 0]
                 ["output" "results"]
                 ["output_mode" "xml"]]]
    doall (into {} (for [[k v] ruleset] [k (if (.containsKey opts k) (.get opts k) v)])))))
```

Now that I have my argument map, I need to process it into Args objects understood by the Splunk Service class. I don't know about you, but this all feels just a bit fiddly:

5b ⟨*build the queryargs object* 5b⟩≡ (9a)

```
(defn build-splunk-queryargs [argument-map]
  (let [rulelist ["earliest_time" "field_list" "latest_time" "status_buckets"]
        qa (Args.)]
    (doseq [fieldname rulelist]
      (if (argument-map fieldname)
        (.put args fieldname (argument-map fieldname))))
    args))
```

Later, I have to build the output Args object, which the Splunk Service uses to to configure the output. Obviously. It's the exact same code, and it wasn't cut and paste. I'm thinking this needs an abstraction.

5c ⟨*build the output args object* 5c⟩≡ (9a)

```
(defn build-splunk-output-args [argument-map]
  (let [rulelist ["count" "offset" "output_mode"]
        args (Args.)]
    (doseq [fieldname rulelist]
      (if (argument-map fieldname)
        (.put args fieldname (argument-map fieldname))))
    args))
```

Now that we have everything, *le sigh*, it's time to pass it all to the server. I don't even care much about the Service object once I've instantiated it. I'm using it for a single query, a single Job on the server. And it's a bit redundant to pass both the command and the argument map, but I'm using the map to configure other program behaviors, so it stays as a separate copy of the command content. Between the double-dot argument and the chain operators, I'm seeing a lot of Haskellian inspiration in Clojure.

6a ⟨*send the query to the server* 6a⟩≡ (9a)

```
(defn build-splunk-job [command argument-map]
  (let [queryargs (build-splunk-queryargs argument-map)
        service (Service/connect (.opts command))
        job (.. service (getJobs) (create (first (.args command)) queryargs))]
    (while (not (.isDone job))
      (if (argument-map "verbose")
        (println (format "\n%03.1f%% done - %d scanned - %d matched - %d results"
                         (* (.getDoneProgress job) 100.0)
                         (.getScanCount job)
                         (.getEventCount job)
                         (.getResultCount job))))
      (Thread/sleep 1000))
    job))
```

There are a number of different things we can get from the server. The correct setting is usually "preview", meaning "show me any valid data you've collected, even if it's not complete." Preview will return everything even if the job *is* complete, so it's safe to use at all times. But you can see the options below. Here, using the argument map and the instructions to build the output args object, I ask for a stream (an actual Java `InputStream`) from the server of the data I want:

6b ⟨*request a stream from the server* 6b⟩≡ (9a)

```
(defn get-splunk-stream [job argument-map]
  (let [outputargs (build-splunk-output-args argument-map)
        output (argument-map "output")]
    (case output
      "results" (.getResults job outputargs)
      "preview" (.getPreview job outputargs)
      "searchlog" (.getSearchLog job outputargs)
      "summary" (.getSummary job outputargs)
      "timeline" (.getTimeline job outputargs)
      )))
```

There are two kinds of read operations: we could use the Splunk ResultsReader, which parses the content for you and turns it into a hashmap of keys and values, or you can just get the raw data. I'm going to deal with the readers first.

Instantiating a reader irked me. I could not figure out how to make Java constructors act like first-class objects; I wanted to be able to pick a class and pass it back to the calling function, which could instantiate it on the fly after dereferencing it. I'm sure there are Clojure gurus who can help with that:

7a ⟨*construct a reader of the appropriate type* 7a⟩≡ (9a)

```
(defn construct-reader [stream output-mode]
  (case output-mode
    "xml" (ResultsReaderXml. stream)
    "json" (ResultsReaderJson. stream)
    "csv" (ResultsReaderCsv. stream)))
```

This returns a reader-ready function to print out content based upon the output mode. The `when` macro for handling loops was a real eye-opener.

7b ⟨*make a stream reader with a mode parser* 7b⟩≡ (9a)

```
(defn get-streamtype-reader [output-mode]
  (fn [stream]
    (with-open [reader (construct-reader stream output-mode)]
      (loop [e (.getNextEvent reader)]
        (when (not= e nil)
          (println "EVENT:********")
          (doseq [k (seq (.keySet e))]
            (printf "%s --> %s\n" k (.get e k)))
          (recur (.getNextEvent reader)))))))
```

This returns a reader-ready function to print out the raw content.

7c ⟨*make a generic stream reader* 7c⟩≡ (9a)

```
(defn generic-reader []
  (fn [stream]
    (with-open [reader (InputStreamReader. stream "UTF8")
                writer (OutputStreamWriter. System/out)]
      (try
        (let [buffer (char-array 1024)]
          (while true
            (let [count (.read reader buffer)]
              (if (== count -1) (throw (Exception. "EOF")))
              (.write writer buffer 0 count))))
        (catch Exception e nil)))))
```

I'm completely sure that there's a better way to acheive the symmetry I wanted, and that the function that takes no argument above in "make a generic stream reader" is completely gratuitous, but I kinda liked the way this one came out.

8a ⟨*get the stream reader* 8a⟩≡ (9a)

```
(defn get-reader [command argument-map]
  (let [use-reader (.. command opts (containsKey "reader"))]
    (if use-reader
      (get-streamtype-reader (argument-map "output_mode"))
      (generic-reader))))
```

This has to reveal just how bleedingly new I am to Clojure, because I've never seen a "let cascade" like this before in anyone else's code. But it works!

8b ⟨*main* 8b⟩≡ (9a)

```
(defn -main[& args]
  (let [command (build-splunk-command args)]
    (let [argument-map (build-argument-map command)]
      (let [job (build-splunk-job command argument-map)]
        (let [stream (get-splunk-stream job argument-map)]
          (let [reader (get-reader command argument-map)]
            (reader stream)))))))
```

And the entire program ends up looking like this:

9a ⟨*core.clj* 9a⟩≡

⟨*create the namespace* 3b⟩

⟨*parse command arguments* 4⟩

⟨*build the argument map* (never defined)⟩

⟨*build the queryargs object* 5b⟩

⟨*build the output args object* 5c⟩

⟨*send the query to the server* 6a⟩

⟨*request a stream from the server* 6b⟩

⟨*construct a reader of the appropriate type* 7a⟩

⟨*make a stream reader with a mode parser* 7b⟩

⟨*make a generic stream reader* 7c⟩

⟨*get the stream reader* 8a⟩

⟨*main* 8b⟩

And that's it.

You can now run the program with lein:

9b ⟨*command line* 9b⟩≡

```
lein run -output_mode csv "search magnitude > 4.5"
```

Assuming you have something in your Splunk database with a "magnitude," you should get a CSV dump of all the fields related to it.

Now you know how to talk to the basic Splunk service using Clojure. I'm sure if you're comfortable with Clojure none of the Java access weirdness surprised you at all, but for me this was a pretty good exercise. At times I felt like I was writing in a "fantasy LISP", a Lisp that actually, you know, had *real world* applicability, and that should have done the things I would have expected of a LISP. That fantasy LISP was pretty close to the real deal; it only took me a few hours of `lein run` sessions to knock out all the bugs.

1.6 Index