jcom: A JavaScript Object Shell

DeDominic, Anthony
Eastern Connecticut State University
Willimantic, USA
dedominica@my.easternct.edu

Abstract

With the proliferation of JSON representation of data, shells struggle to make use of them; Currently, shell users rely on tools like jq to handle these structured documents. This paper will discuss an attempt to bring object-like variables to a new shell, jcom. The reason was to allow for shells to natively deserialize json-like documents and to work with them using dot-like notation. This resulted in a cleaner way of working with json-like documents. It also led to less characters being required to solve certain problems. This research suggests shells, like bash, can greatly benefit by providing such structures in later revisions.

1. Introduction

No popular shells, as of late, have the ability to handle modern structured documents, such as JSON. Because of this, it is difficult to use shells for modern text and data handling. This is especially the case for web oriented services; such services heavily rely on transacting structured documents or serialized objects.

This limitation has resulted in the hassle of wrangling various line oriented tools such as the following: sed, grep, awk, xargs, tr, paste, cut, tee, etc, to try and derive value and data. Thus, in a very contrived way, one has effectively wrote a parser. An example of the problem can be seen below[1].

```
# gets an html webpage from duckduckgo,
# processes it into
# "url - page description" lines
# removes ad links
# and only preserves the first three
curl ${SEARCH_ENGINE}$(rawurlencode ${4})
html2 | \
grep -A 2 "@class=result__a" | \
sed '/^--$/d' | \
sed '/@class/d' | \
grep -Po '(?<=\/a(=|\/)).*' | \
paste -d "␣" - - | \
sed 's/\(@href\|b\)=//g' | \
sed '/r\.search\.yahoo\.com/d' | \
head -n 3
```

[1] source: https://github.com/GeneralUnRest/neo8ball-irc/blob/master/lib/search.sh

1.1. Related Works

Object Oriented shells are not an entirely new concept. Many projects, some more serious than others, attempted to solve this data-wrangling issue [1] [2].

1.1.1. shcaml

Shells have very powerful uses; this is why most systems use a shell as the most basic UI for interacting with a operating system. Sometimes it's ideal to use these features to handle interprocess communication. This is also why many programming languages provide the ability to spawn shells–functions like system(), in the standard library. The power of the shell was not lost to some OCaml–a functional language, developers; these developers created, shcaml. This library offered a more powerful shell spawning construct for the OCaml language. One such feature of the library gave the user the ability to utilize OCaml functions as a go-between two programs. This allowed for powerful data-wrangling functional compositions that could take in data from shells and output or exchange structured documents [1].

1.1.2. Named Pipe Shells

Another way of attaining structured inputs and outputs is through an object system. One attempt at object oriented shells attempted to make use of named pipes and file descriptors to mimic objects [2]. This resulted in a new process for every object; this comes with great performance cost. Each object still has limitations in terms of functionality. They are basically just another shell with its own encapsulated data. It also incurs

security risks since processes can write and listen to the objects' named pipes.

1.1.3. Lisp Shells

Lisp–another functional like language, shells have been other ways of trying to add ; examples are like Scheme Shell, and other toy, academic ones [1][3]. The advantage lisp gives over other languages, which makes it suitable for shells, is that the program and data have the same form [3]; this comes in handy in meta programming, where programs can dynamically change to handle certain data. It is also a closed language, as in, one can pull from data that is in an outer function. Closure property is what gives lisp its ability to resemble structured documents. In a way, one can make a lisp-like dialect using JSON.

1.1.4. Powershell

Projects like Powershell are built on the .NET runtime and can thus utilize all of the language features therein. The streaming pipelines in Powershell are merely .NET classes which allow for some structural correctness and parsing [1][4]. As a result, Microsoft prefers to call more than just a shell, but a whole "automation and configuration management framework." One that is capable of handling structured data such as JSON, XML, CSV, etc, REST APIs, and object models.

1.1.5. Ansible

Tools like Ansible, are not quite shells, but are domain specific languages which solve similar problems. Ansible and Salt Stack use code generation and a strongly structured document–"playbook", to construct python code. Since python is a general purpose language, it handles structured and typed data much better than shell code. Ansible as a result has become a cornerstone of modern IT automation, provisioning and configuration management; many of these problems require working with such data.

The next few sections will cover some background on important topics and issues. It is necessary to convey the importance of the project as a whole.

1.2. Service-oriented Architecture

Service-oriented Architectures, or SOA, are modern way to construct, highly available, data rich web applications. In short, a service oriented architecture is a way for applications to share state and provide services over middleware and structured, serialized objects. [5] One way of transacting states between services is through Representational state transfer, or REST. With REST, services are able to share well structured data through document structures like JSON, XML and many others. [6]

POSIX-shells are nearly completely useless for interacting with SOAs. Generally when dealing with web APIs it is recommended to build a general purpose language script instead; If an ansible module exists, that may be another alley of interest.

1.3. Designing and Defining a Domain Languages

Designing domain specific languages is a very slow and arduous task. Some have suggested designing and developing domain specific languages using meta-models; basically a model, modeling models [7]. This process makes use of advanced features of UML version 2.1. This also has the added benefit of code generation.

Defining a language is ultimately the process of creating formal grammars. Generally this is done empirically, as in from experience and trial and error. Some have used machine learning and bottom-up CYK based parsers to generate grammar [8]. This would be useful for languages built using example strings.

1.4. Events

Some system events may impact other systems. Currently, other than using triggers, there is no way to define listen events in POSIX shells. Powershell offers such functionality through the power of its .NET classes [4]. To accomplish a similar function, one must use a blocking statement like this:

```
# file descriptor is needed
# for performance reasons
exec <>99 /some/named-pipe
# consumes a process, must be backgrounded
while read -r line; do echo $line; done <&99 &
echo "event" >&99
```

Given that package management tools used in many linux distributions use shell like scripting, and have a post-install process, it may be helpful for a user to define his own post-install event handlers or notifiers.

1.5. Portability Concerns

Shells are the primary interface into a POSIX system. As a result, many tools and utilities rely on shells to handle and process files and their data.

Even using different coreutils can lead to issues. For instance, many of the GNU coreutils–such as their implementation of grep and df, add features not strictly standardized by POSIX. An example:

```
grep -P '(?>=)somepat' file
# does not work on macOS, which uses,
# mostly, BSD coreutils
```

Different operating systems–ranging from GNU-based Linux distributions, The BSDs and other non-free systems like macOS, Solaris, AIX, and other UNIX, can potentially use different coreutils. Thus, this means that shells scripts can depend on behaviors and unstructured output that are not portable.

1.6. Going Further

1.6.1. Interactively

An OCaml library called shcaml added functional combinators called shtreams which allow for transforming byte streams into structured documents [1]. However, it is merely a library. It has no use interactively. This makes it unsuitable as an actual systems shell.

1.6.2 Readability

Some developers have tried to add other structures to shell input, such as Scheme Shell and other implementations [3]. Using functional composition, one can construct a very powerful shell, but it will not be as clear and concise as normal shell syntax. this is because lisp does not allow for pipelining-like syntax. For instance, consider this simple example and scale it out:

```
(|
  (cat somefile)
  (some-command))

cat somefile | some-command
```

1.6.3. Performance

The named-pipe shell would simply be too slow. The need to spawn new processes for each object and dealing with potentially slow named pipe interaction is too heavy [2].

1.7 Objective

Given all the above, the goal is to build a new shell. This shell should be able to execute simple commands and offer basic shell features like pipelining. To differentiate it from other shells, it should be able directly work and manipulate modern data structures and documents like JSON. The shell should be fully interactive, like a normal shell is. The shell should keep with the traditional syntax as much as possible, both for readability concerns, but also for familiarity. Ideally it should be fast, but since it will be written in a scripting language, this might not be fully attainable. To extend the shell's functionality, it is necessary that it have a plugin system. The plugin system will allow for added features and capabilities.

Ultimately, the goal is to see what a shell with objects can do. It is the goal of this research not only to build a shell, but to also observe the positive utility and advantage of a shell.

1.8. Summary of Results

Below, this paper will cover the features of the shell[2]. The paper will discuss how the program stores variables, including objects, or documents. It will also show examples of usage, both manipulating and using variables. Other examples, such as use cases and the plugin system will be demonstrated.

2. Materials and Methodology

In order to make use of the technologies described, first the user must have NodeJS. The version of node being used on the test machine will be the latest–as of this writing, v7.1.0. With NodeJS comes crucial tools like npm. npm is the tool that handles dependencies; The project makes use of such dependencies like pegjs. When downloading the source of the project from github. It is necessary to run npm install to require dependencies.

The project is version controlled using git and is hosted on a remote repository on github[3]. To check out the project, it would be advised to have a git client. npm can also download the latest build of the project given the url in the footer; this is done by typing npm install url.

2.3.1. Libraries

The project makes use of the following libraries. For a more up-to-date list, please consult the package.JSON

[2] note that the features as described are purely conceptual at this stage.

[3] Source: https://github.com/adedomin/jcom

file in the project which lists all the dependencies and their semver version number:

- pegjs - parser generator
- lodash - utility functions for functional array and object handling.
- js-yaml - a yaml parser for javascript, for an example plugin
- various NodeJS built-ins: fs, process, readline, etc.

2.3.2. Testing Envrionment

The machine being used for testing is a Fedora GNU/Linux, release 24, virtual machine running on a Windows 10 Hyper-V hypervisor. This hypervisor has a Core i7-3930k processor and 32GB of memory. The virtual machine has been allocated 2 cores of the 6 physical, with dynamically resizing memory, up to 16GB. To ensure support for the latest features and capabilities, the latest release–as of this writing, v7.1.0, was installed on the machine. Interaction with the machine will be done over a network; this is accomplished using a remote shell protocol called secure shell (ssh).

Any and all scripts necessary to reproduce the results will be made available in the github link provided in the footer. Please consult the examples/ directory in the repository.

3. Results

3.1. Objects and Structured Documents

The main object of the project is to process structured documents. To accomplish this, the language will offer a plugin system. Plugins in this context are merely javascript functions which return–or return a value via a callback, javascript objects.

```
// store some input or output buffer
// parsed by a plugin named parseJson
SHELL_VARS[ident] = parseJson(buffer)
// a plugin that uses xpath to select some
// values from an XML document and stores it
SHELL_VARS[ident2] = xpathSelector(buffer, args)
```

Ultimately, the variables are stored in one giant associative arrays. This is a common practice for storing variables. Python objects use dictionaries to keep track of their variables. This simplifies the management of the system. The only limitation is that it can only store un-typed javascript objects, such as Strings, Numbers, Objects and associative arrays (objects without functions).

Accessing Variables are accessed using javascript notations encapsulated in template literal-like syntax. This integrates well with the Javascript virtual machine and is likely cleaner than bash or other shells.

```
// getting an array element from an object
${somevar.somearr[1]}
// bash does not have nested objects
${somevar[1]}
// iterate over every element in an array
${somevar.forEach(var => do something)}
// bash
for var in ${somevar[@]}; do something; done
// NESTED OBJECTS
// not possible in bash as of v4.3
${somevar.someotherobj.somekey.moreobjs.etc}
```

Again, as shown in the above example, javascript notation and variable storage opens up an avenue to handle various kinds of data. This opens up possibilities to accomplish more general tasks that normally is left to general scripting languages like perl, python, ruby or even javascript with Nodejs.

3.2. Events

One feature shells have is the ability to create file descriptors. With these, users have a fast way of connecting programs. File descriptors is usually how one would recreate–what could best be described, as an event loop.

Javascript offers an object called an EventEmitter. With it, one can create low coupled software triggered by events. EventEmitters are very simple, the only two methods that are of importance are on() and emit(). emit() lets one create an event labeled by a string, to be handled by an on(). The on() handles events of a particular label with a function.

In the shell, there exists a simple plugin which leverages this power. The shell would provide a builtin command, *emit* and *on*. The *on* command will take a label and a command as arguments. When executed, the system will listen for the label provided.

As a result, when a user calls an *emit* with the correct label, the *on* will be triggered and the command associated with the on will be ran. Information can be fed through the *emit* that will flow to the *on* using piping.

```
# prints what it is fed
```

[4] This shows how one makes use of variables and the ways to get and set them.

[5] This shows how the event system works by registering commands or daemons to labels and invoking or feeding more data into them using an accompanied emit.

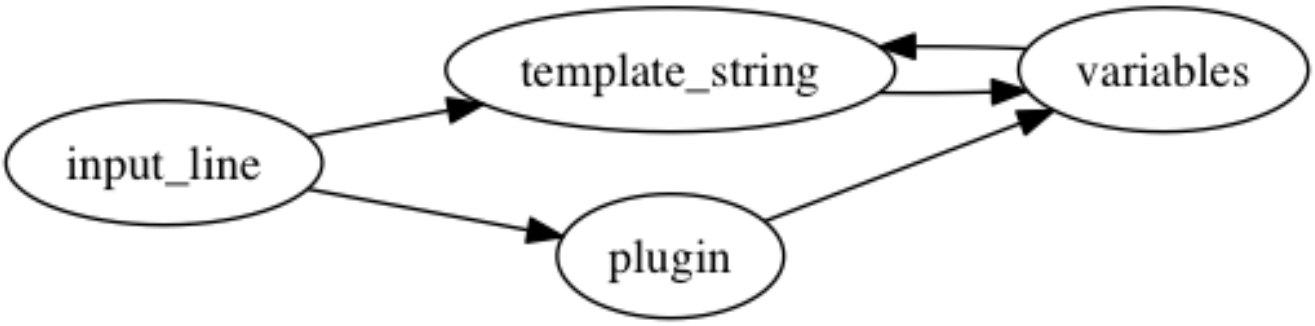

Figure 1: Variable workflow[4]

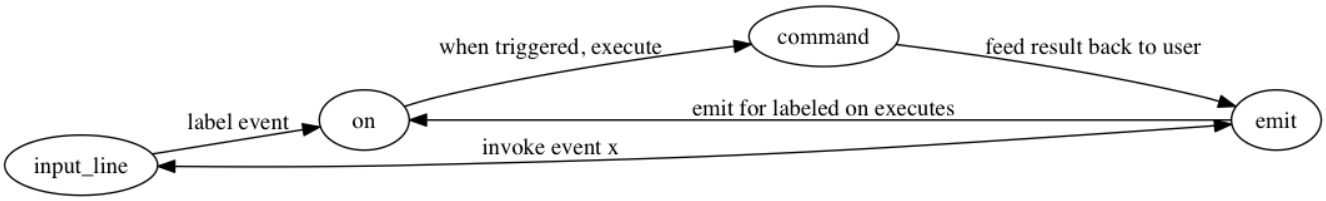

Figure 2: Event workflow[5]

```
on 'echo' cat
# will print hello, world
echo 'hello, ' | emit 'echo'
echo 'world' | emit 'echo'

# reload config on config change
on 'configChange' \
    "kill -SIGWINCH ${some_server_pid}"
# load inf parser plugin
load parse-ini
parse-ini SERVER_CONFIG /etc/server_config
echo "db://newdb.url" \
    | toVar "SERVER_CONFIG.db_url"
parse-ini SERVER_CONFIG \
    --serialize -o /etc/server_config
# reload changed config
emit "configChange"
```

As shown above, these events can be incredibly useful for tasks such as reloading a server when a configuration file is changed. It also demonstrates the power of plugins system. Users can create plugins to parse their configurations into javascript objects. This allows for an easy way to manipulate them and reconstitute them, with changes, back to a file.

3.3. Command and File Parsers

As shown above, the shell can simplify some very common configuration tasks. As suggested, many common shell tasks involve handling file data. To do simple tasks like modifying configuration files can result in humorous stream processing scripts. Similar to how JSON objects have defined schema, many times configuration files have some formal structure. A common structure is the key:value pattern. Usually files have keys and their values and they are separated by a simple equality sign; such files are easily handled by tools like grep or sed.

More advanced configuration structures like ini files, give the user the ability to have single-level nested objects and comments. These files can still be easily transformed with basic line oriented tools. However, consider rich configuration structures like YAML or even Java .properties files. These structures can have infinitely nested objects, arrays and various primitive types. Both of these configuration files have type affinity systems. These become slightly harder to modify with line oriented tools.

By implementing parsers in javascript, a user could then load it into the shell and begin manipulating it using simple object dot notation.

Listing 1: example 1

```
# parse nagios's main configuration
load nagios-parser

nagios-parser --to-var NAGIOS nagios.cfg
 # prints info related to host1
echo ${NAGIOS.host1}
${NAGIOS.host1 = {new obj}} # new host1
nagios-parser --serialize -o nagios.cfg
```

[6]This shows how commands feed serialized documents and how plugins manipulate the shell's variable pool to store them.

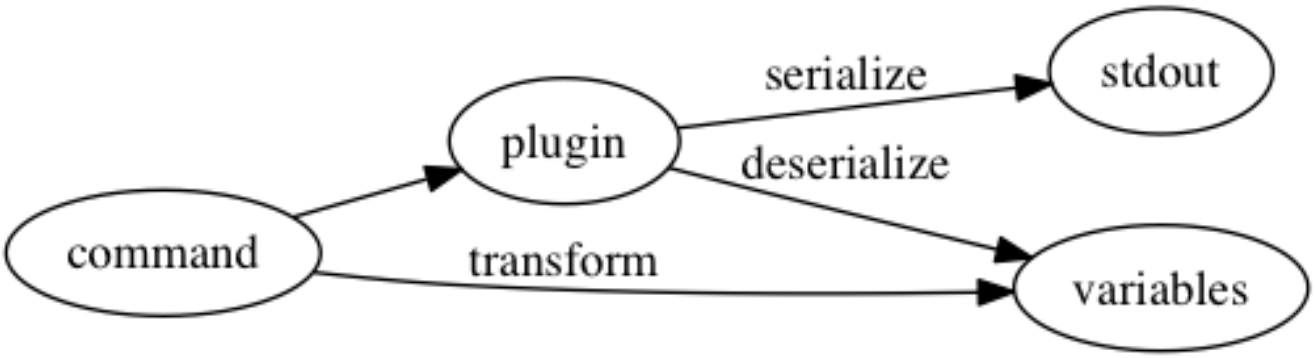

Figure 3: Document parsing and manipulation[6]

All that needs to be done is to make a parser for it. Plugins, in a way, are no different from normal shell programs[7].

Citations

[1] A. Heller and J. Tov, "Caml-Shcaml: An OCaml Library for UNIX Shell Programming," 2008. DOI: 10.1145/1411304.1411316

[2] J. Haemer, "A New Object-Oriented Programming Language: sh," 1994 [Online]. Available: http://https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/haemer.ps

[3] J. Ellis, "A LISP SHELL," 1980. DOI: 10.1145/947639.947642

[4] J. Snover, "Monad Manifesto." Microsoft, Aug-2002 [Online]. Available: http://www.jsnover.com/Docs/MonadManifesto.pdf

[5] Perrey and Lycett, "Service-oriented architecture." IEEE, Jan-2003. DOI: 10.1109/SAINTW.2003.1210138

[6] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with representational state transfer (REST)." Elsevier, 2008. DOI: 10.1016/j.websem.2007.11.002

[7] B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML." IEEE, May-2007. DOI: 10.1109/ISORC.2007.10

[8] K. Nakamura and T. Ishiwata, "Synthesizing Context Free Grammars from Sample Strings Based on Inductive CYK Algorithm." Springer, Grammatical Inference: Algorithms and Applications, 2000. DOI: 10.1007/978-3-540-45257-7_15

[7] Discussion component of the paper is pending further results, to be written.